A systematic literature review on bad smells – 5 W's: which, when, what, who, where

Elder Vicente de Paulo Sobrinho, Andrea De Lucia, Marcelo de Almeida Maia

Abstract—Bad smells are sub-optimal code structures that may represent problems needing attention. We conduct an extensive literature review on bad smells relying on a large body of knowledge from 1990 to 2017. We show that some smells are much more studied in the literature than others, and also that some of them are intrinsically inter-related (*which*). We give a perspective on how the research has been driven across time (*when*). In particular, while the interest in duplicated code emerged before the reference publications by Fowler and Beck and by Brown et al., other types of bad smells only started to be studied after these seminal publications, with an increasing trend in the last decade. We analyzed aims, findings, and respective experimental settings, and observed that the variability of these elements may be responsible for some apparently contradictory findings on bad smells (*what*). Moreover, we could observe that, in general, papers tend to study different types of smells at once. However, only a small percentage of those papers actually investigate possible relations between the respective smells (co-studies), i.e., each smell tends to be studied in isolation. Despite of a few relations between some types of bad smells have been investigated, there are other possible relations for further investigation. We also report that authors have different levels of interest in the subject, some of them publishing sporadically and others continuously (*who*). We observed that scientific connections are ruled by a large "small world" connected graph among researchers and several small disconnected graphs. We also found that the communities studying duplicated code and other types of bad smells are largely separated. Finally, we observed that some venues are more likely to disseminate knowledge on Duplicate Code (which often is listed as a conference topic on its own), while others have a more balanced distribution among other smells (*where*). Finally, we provide a discussion on future directions for bad smell research.

Index Terms—Software maintenance, reengineering, bad smell

1 Introduction

Software systems need to evolve continuously to cope with new requirements and environment changes. High quality source code plays an important role in this context because the code itself needs to be easy to understand, analyze, change, maintain, and reuse [1]. However, software developers eventually produce sub-optimal code, possibly introducing design problems, i.e., they produce code structures that violate fundamental principles in software engineering, such as high cohesion and low coupling. This situation is also known as "*technical debt*" [2], a debt that developers have with the system organization. In the short-term, technical debts may bring benefits, such as, higher productivity or shorter release time, but in the long-term debts can cause a significant amount of extra work.

Some studies have reported on the negative impact of bad smells in software maintenance. Aiko and Leon [328] report that 27% of maintenance problems are related to bad smells. On the design side, relationships between architecture problems and bad smells have also been reported [318, 320]. Gurp and Bosch [3] report that architectural problems could cause software discontinuity or loss of hegemony. Bad smells could be used as indicators for low code quality code, representing potential threats [4].

On the other hand, bad smells may not be as harmful as generally claimed. Rahman et al. [192] report that bugs are not significantly associated to duplicated code. Also, in some situations, writing code with the presence of bad smells may be even the best option for developers [387].

Although a huge body of knowledge has been produced over almost 30 years, it still lacks more organization. So, this paper aims at elaborating a systematic literature review on bad smells between the years 1990 and 2017 taking into consideration the diversity of different kinds of bad smells (**which**); the evolution of the interest of researchers in bad smells (**when**); the aims, findings, and material for experimental setup (**what**); the different people and groups interested in bad smells (**who**); the distribution of papers among venues (**where**).

Although there are already some surveys on bad smells, they do not cover the literature in a comprehensive way. Differently from previous work [5, 6, 7, 235], we propose a methodology aimed at minimizing the possibility of missing relevant papers. As a result, unlike Zhang et al. [7] we show that the research interest (measured by number of papers) on bad smells has increased over the years and many papers have recently studied the impact of bad smells.

Another distinguishing characteristic of our work is the investigation of the co-occurrence of different bad smells in the same paper. We will show that while Duplicate Code is widely studied, there are only a few studies that consider the combination of Duplicate Code with other kinds of bad smells, confirming on a much larger scale

- *Elder Vicente is with Federal University of Triângulo Mineiro, Department of Electrical Engineering, Uberaba, Brazil. e-mail: elder.sobrinho@uftm.edu.br.*
- *Andrea De Lucia is with University of Salerno, Fisciano (SA), Italy, e-mail: adelucia@unisa.it.*
- *Marcelo A. Maia is with Federal University of Uberlândia, Faculty of Computing, Uberlândia, Brazil, e-mail: marcelo.maia@ufu.br.*

the results obtained by Zhang et al. [7]. Moreover, if we disregard those papers that study only DUPLICATE CODE, we observe that the other kinds of bad smells are typically studied with other smells (e.g., a paper investigating LARGE CLASS and FEATURE ENVY). That kind of co-occurrence of bad smells in papers has not been addressed in previous reviews. Moreover, the co-occurrence of smells in the same paper does not imply that any inter-relationship between them is investigated. We further investigate this matter, reporting on smells that are co-studied for investigation on the inter-relationship raised by their co-occurrence in code. Considering this difference between papers about DUPLICATED CODE and papers about other types of bad smells and considering that there are previous surveys considering only papers about DUPLICATED CODE [6, 235], we observe that DUPLICATED CODE can be considered as a topic on its own deserving a different and more specific systematic literature review, with respect to the one presented in this paper. For this reason, in some RQs, we do not consider papers discussing DUPLICATED CODE, and only focus on the other types of smells, in particular the RQs concerning the thematic area *what*. We consider papers concerning DUPLICATED CODE in RQs where we want to emphasize the difference with papers about other smell types. Also, we will refer to papers studying only DUPLICATED CODE as *Duplicated Code Group* (DCG), whereas we will refer to papers studying other types of bad smells as *Other Bad Smell Group* (OBSG).

The main contributions of this paper are:

- a large-scale review on bad smells, larger than previous surveys;
- a systematic organization of the findings on bad smells other than DUPLICATED CODE produced over more than a decade in more than a hundred papers, to unveil the convergent, divergent and main findings;
- a interpretation of the reported findings, with further anecdotal evidence obtained from the consolidated results;
- several statistical descriptions of data to characterize the area, including a clear characterization of collaborations among researchers using Social Network Analysis;
- avenues for further work which would help the community, either experts or newcomers to this field.

The remainder of this paper is structured as follows: in Section 2 we present background information and discuss related work. In Section 3 we present our research questions, defined with respect to five thematic areas (TAs): **which**, **when**, **what**, **who** and **where**. In Section 4, we present the systematic literature review methodology and discuss its limitations and threats. Sections 5, 6, 7, 8 and 9 present the results of our research questions for the five thematic areas. In Section 10 we discuss the main findings of the study and outline some future challenges and research directions. Conclusions are drawn in Section 11.

2 BACKGROUND AND RELATED WORK

This section provides the background for the bad smell research topic. First, we introduce the concept of bad smell. Finally, Subsection 2.2 focuses on previous systematic literature reviews on bad smells.

2.1 Background

Several terms have been used to refer to sub-optimal code: code anomaly, bad smell, code smell, anti-pattern, design smell [8, 317, 320, 366].

Bad smell or *code smell* are terms used by Fowler and Beck [9] referring to sub-optimal code structures that may cause undesired or even harmful effects. These structures were not intended to be formally defined, because “no set of metrics rivals human intuition” [9]. Nonetheless, they defined 22 code structures that could be detected in code and assessed if worthwhile to refactor. Smell detection tools use thresholds on metrics or ad-hoc rules to identify such structures in code, at the price of some inaccuracy.

One well-known example of a bad smell is LARGE CLASS, sometimes also referred to as GOD CLASS or BLOB [312, 371]. This smell is related to classes that are “trying to do too much", whose symptoms may be represented by too many instance variables.

DUPLICATE CODE or CLONE is another example of bad smell, which consists of two or more similar code segments, considering a similarity criterion [235].

Brown et al. [10] define the term *anti-pattern* as a commonly occurring solution to a problem that definitely generates negative consequences. They organize the taxonomy of anti-patterns in three major classes that represent the viewpoints of software developers, software architects, and software managers:

1) Development anti-patterns describe situations encountered by the developers when solving programming problems. These patterns are related to Fowler’s bad smells because their symptoms are more likely to be reflected in the code, such as, BLOB, SPAGHETTI CODE or LAVA FLOW (DEAD CODE). However, the anti-pattern MUSHROOM MANAGEMENT is related to the situation developers face when they are uncertain of requirements and there is no effective way to obtain clarification. The anti-pattern MUSHROOM MANAGEMENT has no direct relationship with a bad smell that can be observed in code.
2) Architectural anti-patterns focus on common problems in system structure. These anti-patterns focus on problems and mistakes in the creation, implementation and management of the architecture, and most of them are not directly reflected in the code such, as VENDOR LOCK-IN, where systems are highly dependent on proprietary architectures. Nonetheless, there are some architectural anti-patterns that can be detected from source code. An example is SWISS ARMY KNIFE that is manifested by a class that has an excessively complex interface.
3) Management anti-patterns are related to human communication and people issues. They are related to the software process and are not necessarily directly observable in source code.

Bad smells described by Fowler and Beck [9] emerge from local problems in code, i.e., they are represented by low-level programming structures, whereas only part of

the anti-patterns described by Brown et al. [10] can be observed in code considering the developers' or architects' viewpoints.

Because there is some intersection between these two concepts, the terms bad smell and anti-pattern are used as synonyms [11]. Other studies distinguish these terms: a bad smell represents something "probably wrong" in the code, while an anti-pattern is a design problem in the source code, because it results from the application of a wrong solution to a recurring problem [333]. In other words, a bad smell is a symptom of a bad design choice, so it might indicate the application of an anti-pattern.

The context of our systematic literature review is concerned with the analysis of source code to identify potential design problems (i.e., bad code smells). For these reasons, in our systematic literature review, we use the term *bad smell* to refer to *a symptom of a bad decision observed in a program's low-level structure*.

2.2 Related Work

Given the large extension of the *bad smells*, evidenced by the high number of studies conducted over decades, some surveys/reviews have already been presented. We list below an overview of relevant surveys related to bad smells.

Rattan et al. [235] presented a systematic literature review on DUPLICATE CODE aimed at identifying detection techniques and tools. In addition, papers were classified and tools were compared. The article database was constructed based on the query string *"clone"* on the following sites: IEEExplore, ACM DL, ScienceDirect, Springer, and Wiley.

Bandi et al. [5] aimed at answering which techniques and metrics have been empirically evaluated within a gradual code decay process, which negatively affects software quality. They report that coupling metrics are widely used for code decay detection. The article database was constructed using textual search where the query was composed of several terms and logical operators in the following sites: IEEExplore, ACM DL, Scopus and Google Scholar. Their work has a wider scope because it includes not only the bad smell concept, but architectural and design rule violations as well.

Pate et al. [6] conducted a systematic literature review covering the used methods, the encountered patterns, and the evolution of DUPLICATE CODE. They studied 30 papers (obtained using a procedure similar to the one used by Bandi et al. [5]), and observed that some papers derived conclusions on developers' behavior or intention based only on data analysis from source code. The authors indicate the necessity of empirical studies also involving developers. The study also reports that *"there are contradictions among the reported findings, particularly regarding the lifetimes of clone lineages"* [6].

Rasool and Arshad [417] reports a review regarding techniques used to handle bad smells. Their review includes publications between the years 1999 and 2015. Their results show that the interest of researchers on bad smell detection remained high in the years 2005, 2010 and 2013. However, their study is limited only to papers that study the detection of smells proposed by Fowler and Beck [9]. Besides papers focusing on bad smell detection, we also include papers that have other aims (e.g., Prediction). These differences can produce a more realistic result concerning the interest on bad smells. Rasool and Arshad [417] also report on the smells defined by Fowler that the tools are able to detect. However, they did not analyze the possible association between the classification of tools (e.g., research prototype, commercial, public) and the smells that they handle, e.g., the tools classified as research prototypes are oriented toward handling less common smells (emerging smells). Fernandes et al. [12] present a systematic literature review of bad smell detection tools, but they also have similar limitations as above.

Vale et al. [13] present a systematic literature review about bad smells in the software product line (SPL). Their selection strategy is limited to papers that study smells in the context of software product line. Papers concerned with bad smells out of this context were rejected. The authors reviewed 18 papers and reported that research on bad smells in the SPL context is a relatively new topic, starting in 2007. They also identified 70 bad smells classified as 49 code smells, 14 architectural smells, and 7 hybrid smells, providing a catalogue of bad smells. Hybrid smells are defined as one type of bad smells that can be identified combining the idea of one or more architectural smells with one or more code smells. Our systematic literature review was more comprehensive, finding more than 100 smells. Differently from previous work, we also report the main findings of papers and their divergences, pointing out current limitations and challenges.

Zhang et al. [7] conducted a systematic literature review from 2000 to 2009 aiming at answering four questions: (i) Which bad smells have attracted most research attention? (ii) What are the aims of studies on bad smells? (iii) What methods have been used in studies on bad smells?, and (iv) What are the evidence that bad smells indicate problems in code? They selected papers from various journals (JSS, EMSE, IST, JSME, TOSEM and SP&E), which have studied one or more bad smells introduced by Fowler and Beck [9]. The final paper selection included 39 papers. They reported that the bad smells that mostly attracted attention were: DUPLICATE CODE (54%), FEATURE ENVY (31%), REFUSED BEQUEST (28%), DATA CLASS (26%), LONG METHOD (21%) and LARGE CLASS (21%). They also reported that DUPLICATE CODE tends to be studied alone, and it is the most studied because it is simple to understand. Concerning the aims, they found that 49% of the papers aimed at developing tools and methods to detect bad smells, 33% aimed at improving understanding on bad smells, and 15% aimed at developing tools and methods to refactor bad smells. Concerning the methodology, they found that 52% of these studies are empirical, 33% are experimental, 12% are questionnaires/surveys, and 2% non-empirical. Concerning the impact, only 5 out of 39 papers investigated the impact of bad smells. The authors observed that the lack of studies on the impact of bad smells may be explained from a common-sense point of view that the negative impact is obvious, and so, it does not deserve research to find such evidence. However, interestingly, four out of the five studies investigating the impact of code smells showed that not all bad smells have negative impact on code, for instance DUPLICATE CODE can increase reliability, whereas DATA

TABLE 1
Comparison between our literature review and previous studies.

Description	Our Paper	[417]	[13]	[235]	[5]	[6]	[7]
Year of publication.	—	2015	2014	2013	2013	2011	2010
Paper dedicated to investigating bad smells.	✓	✓	✓	✓		✓	✓
Search engine to finding the papers.	◎	⊕	⊕	⊗	⊕	⊕	⊕
Range of years considered in the papers of dataset.	[1990, 2017]	[1999, 2015]	[2007, 2014]	[1997, 2011]	[−∞, 2013)	[−∞, 2010)	[2000, 2009]
Bad smell scope.	Any	Fowler	Any	DC	Brown, Fowler	DC	Fowler
Number of papers in the dataset.	351	46	18	213	30	30	39
Number of bad smells.	104	22	70	1	6	1	22
Reports the ranking of smells.	✓						✓
Reports the co-occurrence of smells in the papers.	✓						
Reports the birth of smells according to the papers.	✓						
Reports the interest in bad smells over the years.	✓	✓	✓	✓			✓
Reports the aims of the papers.	✓						✓
Reports the main findings of papers.	✓			✓			
Reports which smells are co-studied on the papers.	✓						
Reports the tools used in empirical studies.	✓	✓		✓		✓	
Reports the projects used in empirical studies.	✓		✓	✓		✓	
Reports the research community by types of smells.	✓						
Reports researchers interested to bad smells.	✓						
Reports a social network analysis for authors.	✓						
Reports the propensity of venues to publish papers on a particular set of bad smells.	✓						

✓ Item broached; DC: DUPLICATE CODE; Brown et al. [10]; Fowler and Beck [9].
◎ Manual inspection; ⊗ Simple textual query; ⊕ Complex textual query (uses multiple logical operators).

CLASS, REFUSED BEQUEST, and FEATURE ENVY are not significantly associated with faults.

We observe some issues in those previous surveys that justify our work. Table 1 summarizes the comparison with previous work. None of the presented surveys have the same scale as ours. Moreover, some of them have a scope limited to one specific bad smell, more specifically DUPLICATE CODE [235, 268]. On the other hand, there are comprehensive surveys [5] that investigate papers related to factors that affect progressively and negatively software quality (e.g., bad smells, violations of architecture and design rules). In this case, the high number of factors that affect software quality makes a more comprehensive study impracticable. There are also surveys that investigate only the human perception (developers or researchers) of bad smells [372]. In order to complement this approach, a more detailed review on code smells would be beneficial. From the presented surveys, Zhang et al. [7] study a set of papers with the highest smell diversity. However, the survey is limited only to papers from 2000 to 2009, while most of the papers on bad smells have been published in the last decade. In addition, this survey only considers smells by Fowler and Beck [9], disregarding smells by Brown et al. [10]. Moreover, none of previous literature reviews investigates the relations between different types of bad smells.

3 RESEARCH QUESTIONS

In this section, research questions (RQs) are defined. To ease understanding, we have organized the questions in five *Thematic Areas* (TAs), the 5 W's: (i) Bad smell types (**which**); (ii) Interest on smells over time (**when);** (iii) Aims, findings and settings (**what**); (iv) Researchers (**who**); (v) Distribution of papers among venues (**where)**.

We start with *Which* and *When*, because we want to understand whether there is a difference in the way smells are studied, in particular to understand whether some smells are studied alone or together. We define two different terms related to the fact that different smells occur in the same paper. When different smells are investigated in the same paper, we say they **co-occur** in that paper. Co-occurrence does not necessarily mean that the smells are studied together to investigate some relation between them. In the case where bad smells co-occur in the same paper to intentionally investigate some possible relationship between them, we say that those smells are **co-studied** in that paper. This terminology is used throughout the remainder of this paper. Then, we analyze the *When* part, to see how the interest in the bad smell research topic evolved over the years. Next, we have the *What* part that maps the main findings from the papers on bad smells. Finally, we analyze the structure of collaboration of researchers interested in bad smells (*Who*), along with the venues where the papers have been published (*Where*).

3.1 TA1: Bad Smell Types (which)

This area contains questions intended to summarize information on the types of bad smells that were studied over the period 1990 to 2017.

RQ1.1 Are there bad smells more studied than others (number of papers)? If so, is there any specific reason? Are bad smells studied alone or together with other bad smells (co-occurrences)?
Goal Identifying possible gaps in bad smell research in terms of the coverage of different kinds of smells in the papers. Another gap is the reason for analyzing specific bad smells.

RQ1.2 Has research improved the original catalogs of bad smells? If so, does this improvement occur by the description of unpublished/new bad smells or by the specialization of existing bad smells?
Goal Identifying factors that may have some influence in the definition of new bad smells and if bad smell definitions have changed over time.

3.2 TA2: Interest on Smells Over Time (when)

The purpose of this analysis is to understand whether there is some trend in studies about bad smells. In particular, this area contains questions that investigate the evolution of the interest of researchers on bad smells over time.

RQ2.1 Has the interest in bad smells evolved over the years?
Goal Identifying which bad smells have been studied over the years to unveil possible trends.

RQ2.2 Has the research community interested in bad smells evolved over the years?
Goal Identifying how much the bad smells are valued by the community in the sense of distribution of number of authors over time would indicate relevance and importance in the community.

3.3 TA3: Aims, Findings and Settings (what)

This area presents questions on the primary aims of the paper, what are the main findings on bad smells, and how

experimental execution has been conducted in these studies. As said before, we defined two different terms related to the fact that different smells are considered in the same paper. The co-occurrences are investigated in the *which* thematic area (RQ1.1) to understand the phenomenon from a statistical point of view, co-studies are considered in the *what* thematic area to identify the main findings about the studied relationships (RQ3.3).

RQ3.1 Which are the most commonly targeted aims?
Goal Identifying research opportunities by quantifying the aims and experimental validation techniques. Identifying why some papers are leading to contradictory conclusions with respect to the impact of bad smells, like those presented by Chatterji et al. [268], because results from different objectives may not be directly compared.

RQ3.2 What are the main reported findings?
Goal Identifying the convergent and divergent findings, and highlighting the main contributions on bad smells as well.

RQ3.3 Considering the co-occurrence of bad smells in the papers of our dataset, how many of them actually study some relations between bad smells and what are the main findings of these co-studies?
Goal Understanding the limitations and challenges of research conducted on bad smells, considering a possible relationship derived from co-occurrence in source code of different bad smells. In other words, we investigate those papers that report results on co-studies of smells.

RQ3.4 Which are the most used tools for handling bad smells in the experimental setup?
Goal Helping researchers to choose appropriate tools for dealing with bad smells. Consequently, it may contribute to the development of reproducible work [14] if widely available tools are used.

RQ3.5 Which are the most frequent subject projects used in experimental evaluation?
Goal Helping researchers during the experimental setting design. Furthermore, it may also contribute to the development of reproducible and comparable work.

3.4 TA4: Researchers (who)

This area contains questions linked to who are the prominent researchers, how they relate, how their scientific groups interact, and how the bad smells impacts on the community.

RQ4.1 How is the research community grouped around the types of smells? Do researchers study a broad and diverse set of bad smells, or concentrate on one or a few bad smells?
Goal Understanding how the authors are organized, in particular, considering their interest in different types of bad smells.

RQ4.2 Who are the researchers mostly interested (by number of papers) to the area of bad smells? Which were the countries and universities where bad smells studies have been conducted?
Goal Monitoring and tracking the progress and/or scientific trends through knowledge of who are the main researchers on the bad smell topic. Pointing out where specific kinds of investigation have been carried out to unveil if there are any hubs that concentrate investigation in specific topics.

RQ4.3 How are the authors and their research groups interconnected? Does this interconnection impact on publications?
Goal Understanding how scientific collaboration is established and how that may affect the advance of knowledge. In addition, this information would help to identify the role of each researcher in the scientific community, and thus, complement the process of monitoring scientific progress and/or trends.

3.5 TA5: Distribution of Papers Among Venues (where)

This area contains questions on how research on bad smells has attracted interest among different venues.

RQ5.1 Are there venues more inclined to publish papers on a particular set of bad smells?
Goal Helping researchers to understand which venues have published more papers on bad smells.

4 METHODS

A systematic literature review should follow a formal and reproducible method, which enables the identification, evaluation and interpretation of scientific studies that are related to the desired subject [15]. Therefore, this section aims at establishing the design details, as well as the mechanisms and data that will be used to answer the research questions presented in the previous section.

The execution of this study consists of three main steps: a) definition of the protocol to select and analyze the relevant papers; b) execution of the protocol, and c) result reporting. The protocol consists of the following elements: (i) data extraction from venues: this part shows how to select relevant papers **directly** from predetermined venues; (ii) data extraction from references: to minimize the possibility of some relevant paper not being included in this review, references of the papers selected in the previous step are examined, i.e., papers are selected **indirectly** from the venues; (iii) analysis of the *Final Database*: this part shows the data items that are extracted from the set of all relevant papers. The following subsections describe the elements that constitute the protocol.

4.1 Data Extraction from Venues

In a systematic literature review, the definition of an adequate inclusion strategy can enable the protocol to obtain as many relevant studies as possible [15]. Therefore, this subsection will establish the guidelines used to construct the *Primary Database*, presenting the reasons for the choice of these guidelines.

4.1.1 Limitations of Previous Strategies

Several previous reviews (e.g., [5, 6, 7, 235]) follow the approach by Kitchenham et al. [15, 16]. With some variation, this approach adopts a semi-automatic inclusion strategy defined by the following steps:

1) exploratory enumeration of papers: conducted through a preliminary research aimed at listing a reasonable number of relevant papers;
2) extraction of terms (keywords): conducted through the manual inspection of papers (e.g., Title, Abstract, Keywords), obtained in the previous step. Researchers benefit from their experience to compose a list of terms/words that are strongly linked to the research subject;
3) paper sources: from personal experience of the authors and/or consulting professionals, a list of relevant venues is elaborated (e.g., Conferences, Journals, Workshops);
4) enumeration of papers: generally, the venues obtained from the previous step maintain their publications in online databases, which provide search capabilities. These filters often are logical expressions formed by the terms and their derivations extracted in item 2. Therefore, through these filters, a set of papers is selected to compose the survey.

However, this approach has some drawbacks:

Filter Limitations. The sites ACM, IEEExplore, Sciencedirect, Wiley and Springer maintain in their databases papers published in relevant venues, explaining their usage in surveys [5, 6, 7, 235]. However, suppose that in our design, at step 4, we wanted to retrieve all the papers containing the word *"Clone"* using only the following fields: Title and Keywords. This action would be successful across all sites, except Springer, because it does not provide the option to filter only the desired fields. Therefore, we observed that there is no standard set of filters provided by online databases. This hinders the task of applying the same query within similar fields across all databases, which can produce an undesirable effect of filtering out some relevant papers if the query is not sufficiently general.

Keyword Limitation. Pate et al. [6] use the strategy based on the textual search to retrieve papers dealing with the bad smell DUPLICATE CODE. Then, in the extraction of terms (step 2), they generated a list of words related to the subject and the following logical expression research was constructed (step 4):

> *"(('code' OR 'software' OR 'application') AND ('clone' OR 'cloning' OR 'copy' OR 'duplicate' OR 'duplication' OR 'similarity') AND ('change' OR 'evolution' OR 'genealogy' OR 'maintenance' OR 'management' OR 'tracking'))"* [6].

Our criticism is that the used terms may not be sufficient to capture all relevant papers. Indeed, by analyzing the study by Pate et al. [6], we noted that many terms and their variations, strongly correlated to the subject, were not included, as for example: *"Near-Miss Clones"* [68], *"Cut-and-Paste"* [88], *"Duplicate Code"* [140], *"Code siblings"* [143]. This may limit the quantity and quality of the studies used in systematic literature reviews that use traditional text retrieval provided by the sites (e.g., locating the documents that contain a certain search term [17]), violating the premise that researchers must obtain as many relevant studies as possible [15]. Thus, in the next subsection, we describe our approach to deal with these limitations.

4.1.2 Strategy Adopted in This Paper

We mitigate the previous limitations centralizing the search location and performing **manual inspection** of papers. Thus, our strategy performs the following steps: 1) Selecting paper sources (e.g., venues); 2) Getting all metadata (`BibTeX`) of papers published in venues obtained in the first step to centralize the search location (`Mendeley`[1]); 3) Manually inspecting all papers stored in `Mendeley` and the papers covering the bad smells to be selected for our database.

In order to centralize the search location, we explored the online database of relevant venues. We noted that all of these have the option to export citation files (e.g., `BibTeX`[2]) containing information such as Author, Title, "Venue", Abstract, keywords and others. These fields comprise the data generally used in reviews. However, some sites (e.g., Springer) do not include the "Abstract" field in the reference file. As this information is published on the link of each paper, one can extract this information and include it in the reference file. This procedure can be performed by a utility (e.g., a script) that captures the "Abstract" field. Similarly, one can also automate the process for collecting the reference files. Therefore, with the files, one can use tools (e.g., `Mendeley`) to manage them and/or execute personalized and uniform search operations. With respect to manual inspection, we consider this strategy as the most suitable, because the manual inspection enables the identification of the actual subject of the paper, thus avoiding missing relevant papers.

Next subsection details the selected venues (paper sources) and the reasons why they were included.

4.1.3 Venues

As presented in the previous section, we will inspect each paper published in predetermined venues. This subsection lists these venues, as well as the reason why they were included in the review.

Comparing the list of venues used in reviews/surveys [5, 6, 7, 18, 235], we consider the venues that appeared in at least three of these and have *Topic Area* related to the bad smells. The result was a preliminary list of 14 venues (e.g., ICSE, TSE, WCRE, ASE). We also added some other important venues that appeared in two surveys (ESEC/FSE, FASE, ICSME, OOPSLA, ECOOP and SANER). At the end, our list of venues consists of 20 venues:

ASE - International Conference on Automated Software Engineering; **CSMR** - Conference on Software Maintenance and Reengineering; **ECOOP** - European Conference on Object-Oriented Programming; **EMSE** - Empirical Software Engineering; **ESEC/FSE**[3] - European Software Engineering Conference and International Symposium on Foundations of Software Engineering; **ESEM** - Symposium on Empirical

1. https://www.mendeley.com
2. http://www.bibtex.org
3. Both Joint ESEC/FSE and FSE only

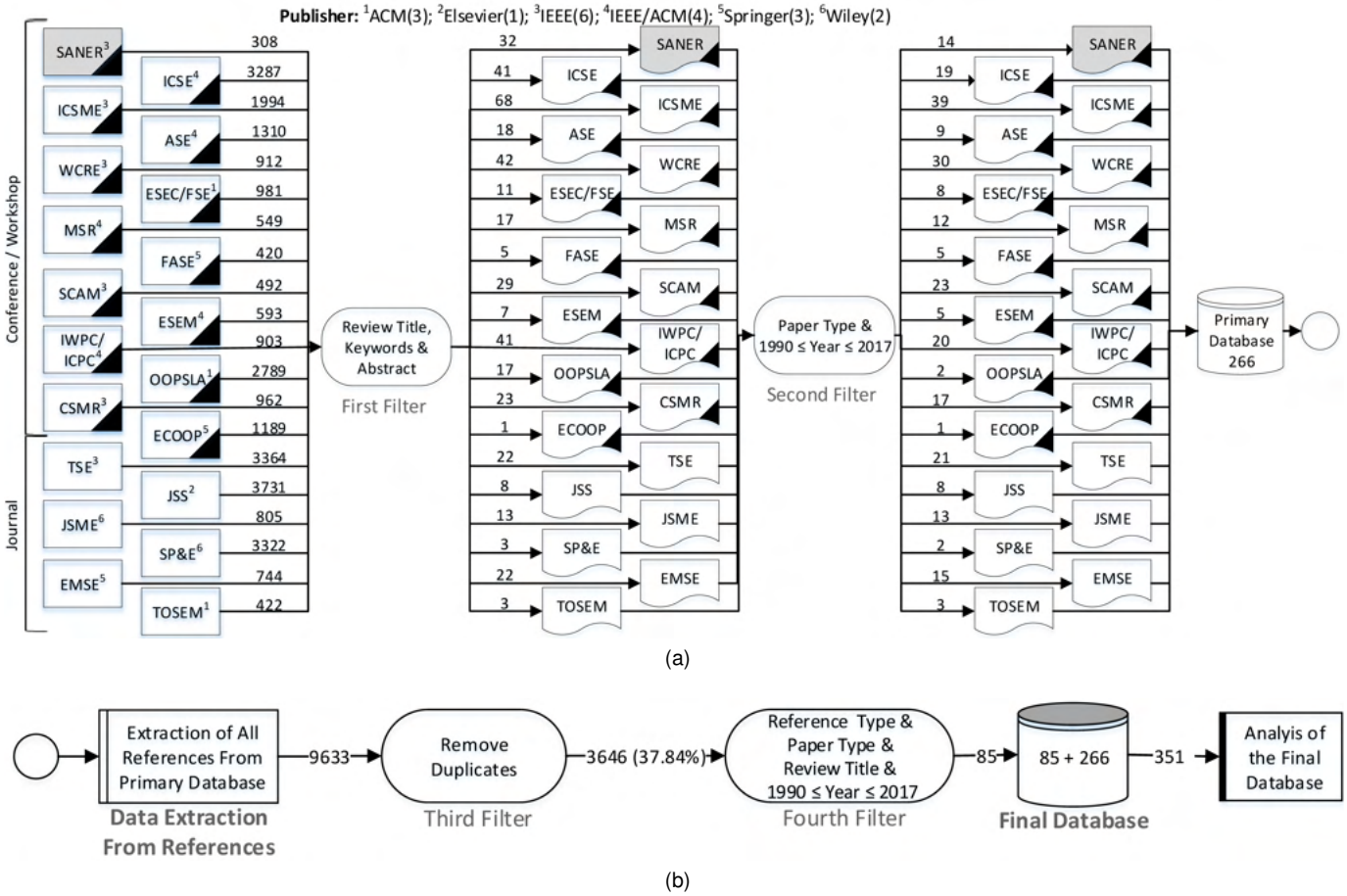

Fig. 1. Protocol Representation. (a) *Primary Database*. (b) *Final Database*.

Software Engineering and Measurement; **FASE** - Fundamental Approaches to Software Engineering; **ICSE** - International Conference on Software Engineering; **ICSME** - International Conference on Software Maintenance and Evolution; **IWPC/ICPC** - International Workshop/Conference on Program Comprehension; **JSME** - Journal of Software Maintenance and Evolution: Research and Practice; **JSS** - Journal of Systems and Software; **MSR** - Working Conference on Mining Software Repositories; **OOPSLA** - Conference on Object-Oriented Programming, Systems, Languages and Applications; **SCAM** - Working Conference on Source Code Analysis and Manipulation; **SP&E** - Software: Practice and Experience; **TOSEM** - ACM Transactions on Software Engineering and Methodology; **TSE** - IEEE Transactions on Software Engineering; **WCRE** - Working Conference on Reverse Engineering. We also consider **SANER** - International Conference on Software Analysis, Evolution and Reengineering, which joined the WCRE and CSMR. Thus, papers published in the joint meeting of WCRE-CSMR (2014) were accounted in the SANER conference.

The considered venues are shown in Fig. 1a enclosed by rectangles. Conferences and Workshops are within the partially filled rectangles and Journals are within the unfilled rectangles. The figure also shows the number of papers found for each venue.

Some venues have their papers published in several indexing sites. For example, for the ASE conference, the 28th edition was published on IEEExplore site and the 29th edition is on the ACM site. For these venues, we consider the papers indexed on both sites. In order to ease the visualization of this information, the legend (*Publisher*) on the top of Fig. 1 allows to verify where the papers of each venue are published.

The next section details the procedure used to select only papers related to the bad smells. These papers form the *Primary Database* (see Fig. 1a).

4.1.4 Primary Database Construction

To mitigate the problem of *Filter Limitations* (see Subsection 4.1.1), we centralized the search site by building a local repository with all the papers from venues listed in Subsection 4.1.3. This repository allows the identification of as many papers as possible related to any software engineering subject. This subsection describes how this repository was used to obtain the papers of the *Primary Database* (see Fig. 1a).

The local repository construction is defined by the following steps:

1) select the venues and find the site that gives access to papers (see Subsection 4.1.3);
2) find the citation files (`BibTeX`) for each paper of all selected venues (see Subsection 4.1.1);
3) complement the citation file with the paper abstract;
4) import citation files to a bibliography management tool (we used `Mendeley`).

Using the centralized repository, we apply two initial filters (see Fig. 1a), which aim at selecting only papers that study the bad smells. In the rest of this subsection, we detail these filters.

First Filter. To mitigate the *Keywords Limitation* problem (see Subsection 4.1.1), for each paper in the repository, we manually read the content of the Title, Abstract and Keywords fields to select those that were related to the bad smells. If these fields are not enough to know whether this paper is about bad smells, then some sections of the paper (e.g., Introduction, Conclusion) are examined. However, applying these criteria to the preliminary set of papers had a challenge: some papers are borderline, for which, inclusion or exclusion from *Primary Database* are disputable. One recurring subject for the borderline papers is refactoring, because refactoring can be used for different purposes (e.g., improve design, readability, internal structure, performance, maintainability and/or comprehension [9]). A reasonable common strategy found in papers is the use of code with bad smells to identify refactoring opportunities [80, 381, 392]. However not all refactoring opportunities are characterized by bad smells [19, 20], some of them can be discovered by generic metrics (e.g., cohesion, coupling) or even through other characteristics (e.g., change-proneness). Therefore, papers that used any type of information, other than bad smells, to find refactoring opportunities were excluded from the *Primary Database*. On the other hand, papers on refactoring operations (e.g., *Extract Method, Move Method,...*) that can be used to remove different bad smells were included (e.g., *Extract Method* is employed to remove DUPLICATED CODE, LONG METHOD, FEATURE ENVY [9]), as well as papers that report techniques that can be used to fix different bad smells for different purposes, such as improving program comprehension. In other words, papers that proposed tools or methods for generic refactoring and do not use it to refactor a specific set of bad smells were excluded from the *Primary Database*.

In general, borderline papers do not discuss or investigate bad smells, they only report a technique/method that can be implicitly used in the context of bad smells. As an example, Tsantalis and Chatzigeorgiou [21] reports an approach to automatically identify *Extract Method* refactoring opportunities, based on identifying two specific situations that occur in code, where those occurrences are not necessarily characterized as bad smells. Another borderline paper [22] proposes an approach to automate the *Extract Class* refactoring based on the analysis of relationships between the methods in a class to identify chains of strongly related methods. However, this paper does not address how the input class is collected, which might not necessarily be affected by bad smells. As a result, the input could be, but not necessarily, classes with low cohesion, which could be, but not necessarily, classes with BLOB [22] or DIVERGENT CHANGE [354].

Thus, in order to clarify our rules, the papers were only included in the *Primary Database* if they met at least one of the following criteria: (i) a paper that reports empirical/qualitative results on bad smells (e.g., detection, analysis, refactoring of bad smells); (ii) a paper that reports a tool or method used to handle bad smells; (iii) a paper that reports the usage of bad smells in closely related domains (e.g., design/documentation). On the other hand, if a paper just mentions bad smells but does not provide further discussion or investigation on the bad smell topic, then it is excluded from the *Primary Database*.

Second Filter. This filter uses two necessary conditions: (i) Paper Type: the paper cannot be a short paper or similar, since these are considered to report preliminary results. Moreover, short papers can be extended and published as full papers. Thus, we select only full research papers, specifically, those published on the main conference track (e.g., *Research Track* of SANER). (ii) Threshold Year: the publication year should be greater or equal to 1990 and lower than April of 2017. The lower bound was defined based on the seminal work on refactoring [23, 24]. The upper bound was defined based on the fact that data collection finished in April of 2017.

After applying the two filters in the centralized repository, the obtained set of papers is referred to as the *Primary Database* (see Fig. 1a), which comprises 266 papers that study the bad smells.

In order to improve the *Primary Database* recall with relevant additional papers that could not be captured with the applied protocol, we proceeded with a snowballing technique: all references cited in the 266 papers in the *Primary Database* were analyzed. We provide details of this protocol in the next section.

4.2 Data Extraction from References

The extraction of references from the papers in the *Primary Database* was automated and non-recursive. We applied two filters to the retrieved references, shown in Fig. 1b:

Third Filter. Removal of references with equal titles;

Fourth Filter. Inclusion of papers with four properties: (i) Reference Type: the reference should be a research paper. Books, technical reports, and similar manuscripts are discarded; (ii) Paper Type: the same criterion as the *Second Filter* (see *Paper Type* on Section 4.1.4) is applied; (iii) Review Title: analogously to the *First Filter* of Section 4.1.4, the papers must study bad smells. However, in this filter, only the paper title is used to manually define its relevance to the theme. If the title was not sufficiently clear about its relationship with the bad smell topic, then that paper was discarded; (iv) Threshold Year: the same criterion as *Second Filter* (see *Threshold Year* on Section 4.1.4) is applied, i.e., papers from 1990 to 2017.

We note here that 9,633 references were extracted and 62.15% (5,987) are duplicated references. In this step, we identified 85 more papers relevant to the bad smells. Interestingly, these 85 papers are from 60 new venues (see Table 2). Considering these new venues, the one that contributed with more papers, contributed with only 5 new papers. Most of the new venues (76.6%) contributed with only one paper.

We constructed the *Final Database* with 266 papers from the *Primary Database* and with 85 papers from the *New Venues Database*.

4.3 Conducting the Protocol and Quality Assessment

Our protocol execution was basically performed by the first author, which has five years of experience in software development, and also has experience in refactoring.

TABLE 2
New venues from references of *Primary Database*.

Venues From References	#
Information and Software Technology (IST)	5
Dagstuhl Seminar Proceedings (DSP)	4
Symposium on Software Metrics (METRICS)	4
Science of Computer Programming (SCP)	4
Journal Automated Software Engineering (JASE)	3
International Conference on Aspect-Oriented Software Development (AOSD)	2
Brazilian Symposium on Software Engineering (SBES)	2
International Conference on Quality Software (QSIC)	2
International Conference on Software Testing, Verification and Validation (ICST)	2
International Symposium on Empirical Software Engineering (ISESE)	2
Practical Aspects of Declarative Languages (PADL)	2
Symposium on Software Visualization (SoftVis)	2
Software Quality Journal	2
Electronic Comm. of the Eur. Assoc. of Software Science and Tech. (ECEASST)	2
Applied Computing Review (ACM ACR)	1
Annual Computer Security Applications Conference (ACSAC)	1
Australasian Computer Science Conference (ACSC)	1
Asia-Pacific Software Engineering Conference (APSEC)	1
Conf. of the Center for Advanced Studies on Collaborative Research (CASCON)	1
Electronic Notes in Theoretical Computer Science (ENTCS)	1
European Conference on Software Architecture (ECSA)	1
Workshop on Evolution of Large Scale Industrial Software Architectures (ELISA)	1
Extreme Programming and Agile Processes in Software Engineering (XP)	1
Frontiers of Software Maintenance (FoSM)	1
IBM Journal of Research and Development (JRD)	1
Innovations in Systems and Software Engineering (ISSE)	1
Intl. Conf. on Mobile Software Engineering and Systems (MOBILESoft)	1
International Conference on Software Engineering and Applications (ICSOFT-EA)	1
Intl. Conf. on Engineering of Complex Computer Systems (ICECCS)	1
Intl. Conf. on Evaluation and Assessment in Software Engineering (EASE)	1
International Symposium on Search Based Software Engineering (SSBSE)	1
International Symposium on Software Testing and Analysis (ISSTA)	1
International Workshop on Emerging Trends in Software Metrics (WETSoM)	1
International Workshop on Principles of Software Evolution (IWPSE)	1
Joint IWPSE-EVOL	1
Conference on Model Driven Engineering Languages and Systems (MODELS)	1
Conf. on Symp. on Opearting Systems Design & Implementation (OSDI)	1
Workshop on Partial Evaluation and Program Manipulation (PEPM)	1
Product Focused Software Process Improvement (PROFES)	1
Quality of Information and Communications Technology (QUATIC)	1
International Workshop on Managing Technical Debt (MTD)	1
International Workshop on Refactoring & Testing (RefTest)	1
International Symposium on Static Analysis (SAS)	1
Brazilian Symp. on Software Components, Architectures and Reuse (SBCARS)	1
Conference on Software Engineering and Knowledge Engineering (SEKE)	1
Workshop on Software Quality and Maintainability (SQM)	1
Intl. Conf. on Testing of Communicating Systems and Intl. Workshop on Formal Approaches to Testing of Software (TestCom-FATES)	1
International Conference on World Wide Web (WWW)	1
Formal Aspects of Computing (FAC)	1
Institution of Engineering and Technology (IET Software)	1
International Journal of Applied Software Technology	1
Journal of Computer Information Systems	1
Journal of Object Technology (JOT)	1
Journal Web Engineering (JWE)	1
Knowledge and Information Systems (KAIS)	1
Computer Science and Statistics (Book Chapter)	1
Computer and Information Science (Book Chapter)	1
Software Evolution (Book Chapter)	1
Java in Academia and Research (Book Chapter)	1
Journal Advances in Computers (Book Chapter)	1

The criteria to manually filter papers related to the bad smells was subject of an agreement analysis. Thus, we randomly selected a sample of 130 papers from the initial set of 29,077 (Note that we consider all papers before the first filter of the protocol, see Fig. 1). In this phase, we do not consider the papers extracted from references in the *Primary Database* (see Subsection 4.2) because in the fourth filter, if the title was not sufficiently clear regarding its relationship with bad smells, then the paper was discarded. This sample has, in equal proportion, papers that were included and excluded from the *Primary Database* and we also included in the sample the borderline papers that were included and excluded from the *Primary Database*. Next, the first and the last authors independently interpreted and applied the inclusion and exclusion criteria to the sample of 130 papers. At the end, we computed Kappa coefficient of agreement [25] for our sample and we arrived at a value of value 0.92, which is characterized as “substantial agreement" according to Landis and Koch [26] and then we could rely on the analysis of the first author only for the rest of the papers.

Moreover, the first and the last author examined the divergences in inclusion/exclusion criteria of papers. In general, we have only five conflicts (3.8% from 130 papers): three false-positives (papers incorrectly included in the topic of bad smells) and two false-negatives (papers incorrectly excluded from the topic of bad smells). For false-positives, actually only one paper was clearly classified incorrectly ([145]) and the others are borderline: one about refactoring [27] and another about static code analysis [28]. In these cases, one author classified the paper as bad smell and the other as non bad smell. However, reading the paper [27], we observe that it proposes a technique to detect refactoring candidates by analyzing method traces and does not discuss the theme of bad smell. Analogously, paper [28] does not discuss the theme of bad smell but the proposed approach could be used in the bad smell context. For false-negatives, we had only two papers. These papers were classified as bad smell by one author and as non bad smell by the other author. However, reading the paper [29], we noted that this identifies refactoring candidates with information about code clones. The last paper is also about refactoring [30] and proposes an approach to classify defects using correction possibilities. This approach takes as input a base of defect examples (bad-designed code, in our context it is a bad smell) with correction (refactoring to fix this bad designed code). At the end, this paper also concludes that the technique to classify defects was able to identify design anomalies (bad smells).

4.4 Analysis of the *Final Database*

In this section, we describe the relevant information that should be extracted from the papers in the *Final Database* in order to answer the research questions in Section 3.

Before describing the extracted fields, we observed that from all papers in the *Final Database*, 227 (64.7%) are related **only** to the bad smell DUPLICATE CODE. The rest of the papers, 124 (35.3%) study a set of bad smells (e.g., DUPLICATE CODE with LARGE CLASS) or study bad smells different from DUPLICATE CODE (e.g., FEATURE ENVY). So, we decided to distinguish between these two sets of papers. We refer to these as ***Duplicate Code Group (DCG)*** and ***Other Bad Smells Group (OBSG)***, respectively. In order to improve the organization, the bibliographic references at the end of the paper (Section *References*) are divided in three parts: the first part lists the papers that do not meet the criteria established by the protocol (e.g., short papers), but were cited somewhere (e.g., Introduction); the second part lists papers classified as DCG; and the third part lists the OBSG papers.

Moreover, after a first analysis of selected papers, we decided that those studies that investigate **only** the bad smell DUPLICATED CODE (DCG) would be analyzed only in the four research questions RQ1.1, RQ2.1, RQ4.1 and RQ5.1

TABLE 3
Data items extracted from each paper and related research questions.

#	Data Item	Description	TAs & RQs
D1	Bad Smell(s)	List of bad smell(s) that the paper study.	TA1: **Which.** RQs: 1.1; 1.2.
D2	Year	In which year was the study published?	TA2: **When.** RQs: 2.1; 2.2.
D3	Purpose(s)	The main objective(s) of paper.	TA3: **What.** RQs: 3.1; 3.2; 3.3; 3.4; 3.5.
D4	Tools	What tools we used to handle the bad smell(s)?	
D5	Projects	Which open source projects were analyzed?	
D6	Author(s)	The author(s) of the paper.	TA4: **Who.** RQs: 4.1; 4.2; 4.3.
D7	Institution(s)	The institution(s) of author(s): University, Company, etc.	
D8	Country(s)	The country of institution(s).	
D9	Venue	Where the paper was published (MSR, ASE, etc.).	TA5: **Where.** RQ: 5.1.

which are in four Thematic Areas, *Which*, *When*, *Who* and *Where*, to highlight differences between studies on DUPLICATED CODE and other smell types. This is due to the fact that: 1) this is a very mature research topic that researchers started to take into account before work by Fowler and Beck [9] and work by Brown et al. [10]; 2) this field has some peculiarities, which deserve a separate more specific study and indeed there are surveys only about this topic (e.g., [6, 235]); 3) as already observed in a previous study [7], this smell tends to be studied alone, whereas one of the goals of our systematic literature review is to analyze whether and why code smells are studied together; 4) we also observed that the community working only on DUPLICATE CODE and the community working on the other types of smells are largely separated.

Table 3 describes the fields that were extracted from the papers. This table also shows the research questions that rely on each field. The fields are described in the following.

4.4.1 Bad Smell Information Field — D1

For each paper, we extracted the list of all kinds of bad smell occurring in papers (D1). We manually inspected the fields Title, Abstract, Keywords and some sections (e.g., Introduction, Conclusion). According to the definitions of bad smells presented in each paper, we grouped the terminology in unique terms. Therefore, from this inspection, we defined the list of all bad smells considered in this work. Some examples of this grouping process are presented in the next paragraph.

According to Fowler and Beck [9], "LARGE CLASS *occurs when a class is trying to do too much, it often shows up as too many instance variables*". Brown et al. [10] defines "BLOB CLASS *is found in designs where one class monopolizes the processing, and other classes primarily encapsulate data*". According to Lanza and Marinescu [31], "GOD CLASS *refers to classes that tend to centralize the intelligence of the system. Performs too much work on its own, delegating only minor details to a set of trivial classes and using the data from other classes*". Certain studies (e.g., [312, 371]) generalize the concepts and consider that the LARGE CLASS is also known as BLOB, WINNEBAGO, and/or GOD CLASS. On the other hand Mäntylä et al. [371], reports that the LARGE CLASS can be detected and analyzed from two points of view: 1) one point related to the measurement of the class size, using traditional metrics (e.g., Lines of Code — LOC)[4]; 2) another point related to the lack of cohesion, i.e., classes that have responsibilities with little or no relationship among them. Some papers [317, 326, 331] study "LARGE CLASS" relating it to size (e.g., LOC) and complexity (e.g., McCabe Cyclomatic [32]). In this case, the notion of complexity — referred to as "COMPLEX CLASS" — is defined as: *"a class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOC"* [317]. The literature also identifies a LARGE CLASS type called BRAIN CLASS, defined as: *"classes tend to be complex and centralize the functionality of the system, but, differently from* GOD CLASSES*, they do not use much data from foreign classes and are slightly more cohesive"* [378]. Some papers [299, 367] even concurrently investigate the concept of LARGE and BRAIN CLASS. In this survey, the term LARGE CLASS is used to group the papers studying the concepts: BLOB, WINNEBAGO, and/or GOD CLASS, even because we did not find papers simultaneously studying a combination of those bad smells. On the other hand, we found some papers studying the following combination: COMPLEX CLASS *vs.* LARGE CLASS; BRAIN CLASS *vs.* LARGE CLASS; LARGE CLASS ONLY *vs.* LARGE CLASS. Therefore, our classification considers the COMPLEX CLASS, BRAIN CLASS, and LARGE CLASS ONLY bad smells separately from the LARGE CLASS.

4.4.2 Time Field — D2

For each paper, we also extracted some field to show how research on the bad smells has attracted interest over time. In this case, the year (D2) was extracted from the citation files.

4.4.3 Empirical Study Related Fields — D3:D5

The aims, findings and implementation aspects covered in the papers were extracted. More specifically, we manually identified the purposes featuring of papers (D3), we also catalogued the tools (D4) and projects (D5) used in experimental setups. For reasons of availability, we catalogued only the data of open source projects.

4.4.4 Identification and Place Fields — D6:D9

In the following, we describe how the identification field was extracted. The author (D6) list aims at reporting the papers of each author. So, it is necessary that each author has a unique name in that list for each paper. Automatic extraction of the authors' names was not possible because they can occur in different forms (e.g., *"Godfrey, Michael W."* [178] and *"Godfrey, M"* [230]). Using last names and first name initials was not possible either, because similar names can exist (e.g., *"Tung Nguyen"* [117] and *"Tien Nguyen"* [151]). So, author names were manually extracted inspecting each title page. Moreover, from that inspection we also extracted other fields such as, email, address, institution, and web page. These elements allow checking if an author has different

4. In Taba et al. [317] this concept is referred as "LARGE CLASS" while in our survey we will use the term "LARGE CLASS ONLY" [311].

name variations, to avoid inconsistencies/duplication in the author database.

For each paper, we also extracted some fields to show how research on bad smells has attracted interest among places. In this case, the fields D7 (Institution) and D8 (Country) were extracted in the same manual inspection to extract Authors. Finally, the field D9 (Venue) was also extracted from the citation files.

4.5 Limitations and Threats to Validity

In order to filter the most relevant piece of work on bad smells, we investigated papers published in a limited set of venues: 14 Conferences and 6 Journals. In this set, not every venue related to software engineering was included. Only those that are widely considered relevant to our subject were considered. Therefore, papers relevant to our subject may have been published in other venues. To mitigate that situation, we also investigated the bibliographic references of the papers included in the *Primary Database*, snowballing for other relevant papers.

The *Final Database* of our survey does not consider every type of study (e.g., Books, Technical Report, Thesis). We do not consider short papers or similar (e.g., papers from the minor conference track). However, our assumption is that if these studies are relevant, there is a high probability that researchers will also publish them in a relevant conference/journal as a full paper.

Another limitation of this survey is that only the authors participated in the selection and analysis of the papers, in particular, the protocol execution was basically performed by the first author and a subset of papers was checked by the third author. However, our Kappa coefficient is characterized as "substantial agreement" (see Subsection 4.3). From this qualitative analysis, we observe that we can not control 100% of all factors. However, discrepancies occur at a very small rate and would not affect the results.

We considered only those papers that explicitly investigate the bad smell issue (see *First Filter* in Subsection 4.1.4). For the sake of scope limitation, papers investigating concepts related to bad smells were not included in this survey if not explicitly concerned with studying bad smells, for instance, studies on refactoring that do not refer to bad smells when proposing refactoring actions were not included.

Indeed, there is still a threat related to this study. The process of paper filtering was based on manual reading of major paper elements as necessary, and indeed, subject to human error. However, the further analysis of the features of the paper mitigates the possibility of false positives being included, but still false negatives may exist.

5 Results on Bad Smell Types (TA1: which)

The following subsections detail the observations on the prevalence and diversity related to the different types of bad smells.

5.1 RQ1.1: Are there bad smells more studied than others (number of papers)? If so, is there any specific reason? Are bad smells studied alone or together with other bad smells (co-occurrences)?

Table 4 shows the entire set of bad smells studied in the *Final Database*. The column *Together* shows the number of papers where these bad smells co-occurred with other smell(s), the column *Alone* shows the number of papers where the bad smell was the only one studied, and the column *Total* is the sum of both. We observe that the bad smells studied the most are: (i) Duplicate Code; (ii) Large Class; (iii) Feature Envy (iv) Long Method, and (v) Data Class.

TABLE 4. Bad smells sorted by number of papers in *Final Database*.

Bad Smells	[1]Together		[2]Alone		Total	
Duplicated Code	18	5.1%	227	64.7%	245	69.8%
Large Class (Blob Class, God Class)	79	22.5%	8	2.3%	87	24.8%
Feature Envy	46	13.1%	3	0.9%	49	14.0%
Long Method (God Method)	47	13.4%	1	0.3%	48	13.7%
Data Class	37	10.5%	0	0.0%	37	10.5%
Shotgun Surgery	32	9.1%	0	0.0%	32	9.1%
Refused Bequest	30	8.5%	0	0.0%	30	8.5%
Long Parameter List	26	7.4%	0	0.0%	26	7.4%
Spaghetti Code	23	6.6%	0	0.0%	23	6.6%
Message Chains Class	18	5.1%	0	0.0%	18	5.1%
Few Methods (Lazy Class, Small Class)	17	4.8%	0	0.0%	17	4.8%
Abstract Class (Speculative Generality)	16	4.6%	0	0.0%	16	4.6%
Function Class (Func. Decomposition)	16	4.6%	0	0.0%	16	4.6%
Data Clumps	15	4.3%	0	0.0%	15	4.3%
Complex Class Only	14	4.0%	0	0.0%	14	4.0%
Swiss Army Knife	14	4.0%	0	0.0%	14	4.0%
Field Public (CDSBP)	13	3.7%	0	0.0%	13	3.7%
Divergent Change	12	3.4%	0	0.0%	12	3.4%
Misplaced Class	12	3.4%	0	0.0%	12	3.4%
Brain Method	9	2.6%	0	0.0%	9	2.6%
Temporary variable, several purposes	9	2.6%	0	0.0%	9	2.6%
Dispersed (Extensive) Coupling	8	2.3%	0	0.0%	8	2.3%
Intensive Coupling	8	2.3%	0	0.0%	8	2.3%
AntiSingleton	7	2.0%	0	0.0%	7	2.0%
Interface Segregation Principle Violation	7	2.0%	0	0.0%	7	2.0%
Switch Statements	7	2.0%	0	0.0%	7	2.0%
Tradition Breaker	7	2.0%	0	0.0%	7	2.0%
Unit Test Smells	1	0.3%	6	1.7%	7	2.0%
Duplicated code in conditional branches	6	1.7%	0	0.0%	6	1.7%
Large Class Only	6	1.7%	0	0.0%	6	1.7%
Schizophrenic class	6	1.7%	0	0.0%	6	1.7%
Use interface instead of implementation	6	1.7%	0	0.0%	6	1.7%
Brain Class	5	1.4%	0	0.0%	5	1.4%
Middle Man	4	1.1%	1	0.3%	5	1.4%
Ambiguous Interface	4	1.1%	0	0.0%	4	1.1%
Inappropriate Intimacy	4	1.1%	0	0.0%	4	1.1%
Parallel Inheritance Hierarchies	4	1.1%	0	0.0%	4	1.1%
Component Concern Overload	3	0.9%	0	0.0%	3	0.9%
Connector Envy	3	0.9%	0	0.0%	3	0.9%
Duplicate Pointcut	3	0.9%	0	0.0%	3	0.9%
God Pointcut	3	0.9%	0	0.0%	3	0.9%
Lexicon Bad Smells	1	0.3%	2	0.6%	3	0.9%
Primitive Obsession	3	0.9%	0	0.0%	3	0.9%
Redundant Pointcut	3	0.9%	0	0.0%	3	0.9%
Scattered Parasitic Functionality	3	0.9%	0	0.0%	3	0.9%
Smells in Android (Specific)	1	0.3%	2	0.6%	3	0.9%
Type Check (State Check)	3	0.9%	0	0.0%	3	0.9%
Anonymous Pointcut	2	0.6%	0	0.0%	2	0.6%
Classes with Different Interfaces	2	0.6%	0	0.0%	2	0.6%
Composition Bloat	2	0.6%	0	0.0%	2	0.6%
Controller Class	2	0.6%	0	0.0%	2	0.6%
Cyclic Dependency	2	0.6%	0	0.0%	2	0.6%
Extraneous Connector	2	0.6%	0	0.0%	2	0.6%
Forced Join Point	2	0.6%	0	0.0%	2	0.6%
God Aspect	2	0.6%	0	0.0%	2	0.6%
Idle Pointcut	2	0.6%	0	0.0%	2	0.6%
Instanceof	2	0.6%	0	0.0%	2	0.6%
Lava Flow (Dead Code)	2	0.6%	0	0.0%	2	0.6%
Lazy Aspect	2	0.6%	0	0.0%	2	0.6%
Linguistic Antipatterns	0	0.0%	2	0.6%	2	0.6%
Low Cohesion Only	2	0.6%	0	0.0%	2	0.6%
Typecasts	2	0.6%	0	0.0%	2	0.6%
Wide Subsystem Interface	2	0.6%	0	0.0%	2	0.6%
Abstract Method Introduction	1	0.3%	0	0.0%	1	0.3%

continued on next column

continued from previous column

Bad Smells	[1]Together		[2]Alone		Total	
Annotation Bundle	0	0.0%	1	0.3%	1	0.3%
BaseClassKnowsDerivedClass	1	0.3%	0	0.0%	1	0.3%
BaseClassShouldBeAbstract	1	0.3%	0	0.0%	1	0.3%
Borrowed Pointcut	1	0.3%	0	0.0%	1	0.3%
Child Class	1	0.3%	0	0.0%	1	0.3%
Class Global Variable	1	0.3%	0	0.0%	1	0.3%
Class One Method	1	0.3%	0	0.0%	1	0.3%
Comments	1	0.3%	0	0.0%	1	0.3%
Distorted Hierarchy	1	0.3%	0	0.0%	1	0.3%
Empty catch blocks	1	0.3%	0	0.0%	1	0.3%
Extraneous Adjacent Connector	1	0.3%	0	0.0%	1	0.3%
Field Private	1	0.3%	0	0.0%	1	0.3%
God Package	1	0.3%	0	0.0%	1	0.3%
Has Children	1	0.3%	0	0.0%	1	0.3%
Incomplete Library Class	1	0.3%	0	0.0%	1	0.3%
Junk Material	1	0.3%	0	0.0%	1	0.3%
Many Attributes	1	0.3%	0	0.0%	1	0.3%
ManyFieldAttributesButNotComplex	1	0.3%	0	0.0%	1	0.3%
Method No Parameter	1	0.3%	0	0.0%	1	0.3%
Multiple Interface	1	0.3%	0	0.0%	1	0.3%
No Inheritance	1	0.3%	0	0.0%	1	0.3%
No Polymorphism	1	0.3%	0	0.0%	1	0.3%
Not Abstract	1	0.3%	0	0.0%	1	0.3%
Not Complex	1	0.3%	0	0.0%	1	0.3%
Obsolete Parameter	1	0.3%	0	0.0%	1	0.3%
One Child Class	1	0.3%	0	0.0%	1	0.3%
Parent Class Provides Protected	1	0.3%	0	0.0%	1	0.3%
Promiscuous Package	1	0.3%	0	0.0%	1	0.3%
Rare Overriding	1	0.3%	0	0.0%	1	0.3%
Simulation of multiple inheritance	1	0.3%	0	0.0%	1	0.3%
Smells in CSS (Specific - DSL)	0	0.0%	1	0.3%	1	0.3%
Smells in JavaScript (Specific - DSL)	0	0.0%	1	0.3%	1	0.3%
Smells in MVC Arq. (Specific)	0	0.0%	1	0.3%	1	0.3%
Smells in Puppet (Specific - DSL)	0	0.0%	1	0.3%	1	0.3%
Two Inheritance	1	0.3%	0	0.0%	1	0.3%
Unused Interface	1	0.3%	0	0.0%	1	0.3%
Useless Class	1	0.3%	0	0.0%	1	0.3%
Useless Field	1	0.3%	0	0.0%	1	0.3%
Useless Method	1	0.3%	0	0.0%	1	0.3%
Various Concerns	1	0.3%	0	0.0%	1	0.3%

[1] Co-occurrence of bad smells on same paper (e.g.: Large Class and Feature Envy).
[2] Single occurrence of bad smell (e.g.: DUPLICATE CODE occur alone on the paper).
DSL: Domain Specific Language.

Table 5 details how these smells co-occur with other smells in the analyzed papers (e.g., LARGE CLASS co-occurred with the LONG METHOD in 41 papers).

DUPLICATE CODE appears on the top of the list, present in 69.8% of the papers. Also note that in 92.6% (227 out of 245) of these cases, DUPLICATE CODE is studied alone, co-occurring with other bad smells only in 18 out of 245 papers, mostly with those that can be detected with size metrics (e.g., LARGE CLASS, LONG METHOD).

LARGE CLASS occurs in 87 (24.8%) papers. Unlike DUPLICATE CODE, in most papers LARGE CLASS co-occurs with other bad smell(s). This fact may be explained by its intrinsic characteristic (e.g., to be related to many responsibilities). We observe that in papers studying LARGE CLASS: 1) 47.1% also consider LONG METHOD; 2) 45.9% consider FEATURE ENVY; 3) 42.5% consider DATA CLASS, and 4) 34.4% consider SHOTGUN SURGERY, which are also the most studied bad smells.

FEATURE ENVY occurs in 39.5% (49) of papers in OBSG, a reduction of 43.6% compared to LARGE CLASS. Analogously to LARGE CLASS, this smell also has rarely been studied alone. The smell FEATURE ENVY co-occurs with other smells, in particular with LARGE CLASS (81.6%) and/or LONG METHOD (63.2%). Moreover, the co-occurrence of FEATURE ENVY and SHOTGUN SURGERY is also significant (55.1%), which may indicate inter-relationship on code, as we show in Section 7.

LONG METHOD occurs in 48 papers, almost the same as FEATURE ENVY. This smell is studied alone just once. As mentioned previously, this fact may be explained by the high correlation of its occurrence with size/volume metrics (e.g., LOC). The main bad smells co-occurring with LONG METHOD are: 1) LARGE CLASS (85.4%), 2) FEATURE ENVY (64.5%), 3) REFUSED BEQUEST (47.9%) and 4) LONG PARAMETER LIST (43.7%).

DATA CLASS always co-occurs with LARGE CLASS and this smell co-occurs with top-4 smells (DUPLICATE CODE, LARGE CLASS, FEATURE ENVY, LONG METHOD). According to Fowler and Beck [9], DATA CLASS are classes which only contain fields and get and set methods for the fields. So, it is natural that this smell co-occurs with others.

In general, we observe that the bad smells studied the most are related to metrics of complexity, size, and volume. Possibly, this may be related to the cognitive perception as reported by Palomba et al. [326]: *"smells related to complex/long source code are generally perceived as an important threat by developers"*. Table 4 also shows which bad smells have received little attention (e.g., GOD PACKAGE, COMMENTS). Note that 70 out of the 104 (67.4%) analyzed bad smells are studied by at most four papers, indicating that most of the proposed bad smells may be lacking more studies.

For the other part of this research question, on the reasons for the higher prevalence of some smells, the analyzed papers justify the choice of studied bad smells using the following reasons, where we highlighted the papers cited in the respective studies:

- the ability of available utility tools (e.g., `InCode`, `DECOR`, `iPlasma`) to handle the bad smells [299, 300, 328] (LARGE CLASS, FEATURE ENVY, LONG METHOD, DATA CLASS, REFUSED BEQUEST, SHOTGUN SURGERY, LONG PARAMETER LIST, SPAGHETTI CODE);
- bad smells should occur with reasonable frequency in the source code of the analyzed systems [317, 319, 325, 327, 328] (LARGE CLASS, FEATURE ENVY, LONG METHOD, DATA CLASS, REFUSED BEQUEST, SHOTGUN SURGERY, LONG PARAMETER LIST, SPAGHETTI CODE, MESSAGE CHAINS, FEW METHODS);
- popularity and diffusion among practitioners. Some papers consider only bad smells perceived as threats by developers [326, 328, 372] (LARGE CLASS, FEATURE ENVY, LONG METHOD, DATA CLASS, REFUSED BEQUEST, SHOTGUN SURGERY, LONG PARAMETER LIST, SPAGHETTI CODE, FEW METHODS);
- representativeness of the respective design problems [311, 319, 325, 326] (LARGE CLASS, FEATURE ENVY, LONG METHOD, DATA CLASS, REFUSED BEQUEST, LONG PARAMETER LIST, SPAGHETTI CODE, MESSAGE CHAINS, FEW METHODS);
- the tradition and/or consolidation in the scientific literature (e.g., Fowler and Beck [9], Brown et al. [10]) [317, 319, 325, 327, 328] (LARGE CLASS, FEATURE ENVY, LONG METHOD, DATA CLASS, REFUSED BEQUEST, SHOTGUN SURGERY, LONG PARAMETER LIST, SPAGHETTI CODE, MESSAGE CHAINS, FEW METHODS);
- the inter-relation of some bad smells can also influence the set of bad smells analyzed in the papers, e.g., studying LARGE CLASS greatly (90.8%) involves considering other bad smells, such as LONG METHOD

TABLE 5
Top five bad smells — map of co-occurrence of smells.

	Duplicated Code (18\245)		Large Class (79\87)		Feature Envy (46\49)		Long Method (47\48)		Data Class (37\37)	
# Papers	Large Class	13	Long Method	41	Large Class	40	Large Class	41	Large Class	37
	Feature Envy	11	Feature Envy	40	Long Method	31	Feature Envy	31	Feature Envy	25
	Long Method	11	Data Class	37	Shotgun Surgery	27	Refused Bequest	23	Shotgun Surgery	22
	Data Class	7	Refused Bequest	30	Data Class	25	Long Parameter List	21	Long Method	18
	Long Parameter List	7	Shotgun Surgery	30	Refused Bequest	20	Data Class	18	Refused Bequest	17
	Refused Bequest	7	Long Parameter List	26	Data Clump	13	Shotgun Surgery	18	Data Clump	12
	Shotgun Surgery	7	Spaghetti Code	23	Long Parameter List	13	Speculative Generality	14	Long Parameter List	9
	Data Clump	6	Message Chains	17	Misplaced Class	12	Message Chains	13	Brain Method	8
	Schizophrenic Class	5	Lazy Class	16	Divergent Change	11	Complex Class Only	12	Misplaced Class	8
	Divergent Change	4	Func. Decomposition	16	Duplicated Code	11	Lazy Class	12	Duplicated Code	7
	...		...		...		...		...	

and/or DATA CLASS (see Table 4). However, this factor is not explicitly mentioned in the literature;
- the conceptual simplicity of smells is another implicit point to increase the interest, e.g., understanding DUPLICATE CODE relies on the concept of similarity. On the other hand, understanding some other bad smells may require the assimilation of various concepts, e.g., modern characterization of SPAGHETTI CODE is linked to multiple structural features (e.g., inheritance, polymorphism) and semantics (e.g., names of classes and methods suggesting procedural programming) [366].

We also investigated the factors that could explain why DUPLICATE CODE is significantly more studied than other types of smells (occurring in 69.8% of papers). In addition to previous factors, the other reasons of the interest on this smell is that it is extremely versatile with several applications. The following items support the assumption of versatility:

- this type of smell can be found at intra- [69, 70, 74] and inter- [73, 143, 175] software designs;
- this type of smell can be considered regardless of the programming language [135, 144, 182, 261] and/or paradigm [178, 215, 232, 245].

Given the peculiarity of this type of smell, the much higher number of papers dealing with DUPLICATE CODE with respect to other types of smells, and the fact that this type of smell is almost always studied alone, unlike the other types of smells, we conjecture that DUPLICATE CODE is a particular case of bad smell which has been largely studied in a different way than other smell types. In other words, research on DUPLICATE CODE represents a topic on its own which deserves a separate and different type of literature review.

The second most recurring smell is LARGE CLASS and according to Kim et al. [33], a refactoring process can increase some measures (e.g., LOC) that are associated with the LARGE CLASS bad smell as follows: *"preferentially refactored modules experience a higher rate of reduction in certain complexity measures, but increase LOC and crosscutting changes more than the rest of modules"*. This suggests that the LARGE CLASS bad smell could be found even in refactored code, indicating that this smell is not trivial to get rid of, thus helping to explain the large interest in it.

Lessons About RQ1.1

We found that some bad smells are much more studied in the literature than others. DUPLICATE CODE is the bad smell studied the most, and interestingly, it is studied alone. The other most prevalent ones (LARGE CLASS, FEATURE ENVY, LONG METHOD) co-occur in papers. Moreover, to a certain degree, the most studied smells are likely to be related to size metrics, e.g., LARGE CLASS and LONG METHOD.

5.2 RQ1.2: Has research improved the original catalogs of bad smells? If so, does this improvement occur by the description of unpublished/new bad smells or by the specialization of existing bad smells?

To answer this question, we tracked the cited references of each OBSG paper searching for papers that defined bad smells concepts.

Our procedure consisted of the following steps:

- For each bad smell in Table 4:
  - For each OBSG paper studying the current bad smell:
    - we recorded the cited references related to the current bad smell (number of occurrences and year).
  - we labeled the most common and earliest reference as the "origin" of the bad smell.

Table 6 summarizes the data from this procedure. It is important to recognize that the listed bad smells were not necessarily coined in the corresponding papers. The papers represent the first studies on the corresponding smell in the group OBSG. For instance, SPAGHETTI CODE dates back at least since late seventies, but the work by Brown et al. [10] was the first one in the OBSG group where this smell occurred.

The bad smells by Brown et al. [10] are present in 27 papers (21.7%). Similarly, but to a greater extent, the bad smells by Fowler and Beck [9] are present in 102 papers (82.2%). Note that all papers that study the bad smells by Brown et al. [10] also study some bad smells by Fowler and Beck [9]. We also observed that out of the 40 anti-patterns presented by Brown et al. [10], only a small fraction (12.5%) has been studied (see Table 6). One possible explanation is that most anti-patterns defined by Brown et al. [10] cannot be considered as bad smells and are difficult to be detected with automated analysis of code or from version control repositories. For example, some of them relate to project management issues, others to architectural bad decisions. Only a third of those anti-patterns relate directly to code and can be actually considered as code smells.

TABLE 6
Origin of bad smells in OBSG papers.

Brown et al. [10] (1998): Spaghetti Code 22; Function Class (Functional Decomposition) 16; Swiss Army Knife 14; Lava Flow (dead code) 2; Controller Class 2.
Fowler and Beck [9] (1999): Large Class (Blob Class, God Class) 84; Feature Envy 48; Long Method (God Method) 46; Data Class 36; Shotgun Surgery 31; Refused Bequest 29; Long Parameter List 25; Duplicated Code 18; Message Chains 18; Abstract Class (Speculative Generality) 15; Few Methods (Lazy Class, Small Class) 15; Data Clumps 14; Field Public (Class Data Should Be Private - CDSBP) 13; Divergent Change 12; Misplaced Class 11; Temporary Variable 8; AntiSingleton 7; Switch Statements 7; Duplicated code in conditional branches 5; Middle Man 5; Inappropriate Intimacy 4; Parallel Inheritance Hierarchies 4; Primitive Obsession 3; Alternative Classes with Different Interfaces 2; Comments 1; Incomplete Library Class 1.
Deursen et al. [34] (2001): Unit Test Smells 7.
Demeyer et al. [35] (2002): Type Check (State Check) 3.
van Emden and Moonen [379] (2002): Instanceof 2; Typecasts 2.
Hannemann and Kiczales [36] (2002): Cyclic Dependency 2.
Marinescu [37] (2002): Brain Class 5.
Martin [38] (2003): Interface Segregation Principle Violation (ISPV) 6; Use interface instead of implementation 5; Wide Subsystem Interface 2; God Package 1.
Mäntylä et al. [371] (2004): Complex Class Only 13; Large Class Only 6.
Monteiro and Fernandes [39] (2005): Lazy Aspect 2.
Lanza and Marinescu [31] (2006): Brain Method 9; Dispersed (Extensive) Coupling 8; Intensive Coupling 8; Tradition Breaker 7.
Piveta et al. [40] (2006): Anonymous Pointcut 2.
Srivisut and Muenchaisri [41] (2007): Duplicate Pointcut 3; Abstract Method Introduction 1; Borrowed Pointcut 1; Junk Material 1; Various Concerns 1.
Trifu and Reupke [42] (2007): Schizophrenic Class 6.
Abebe et al. [43] (2009): Lexicon Bad Smells 3.
Garcia et al. [44] (2009): Ambiguous Interface 4; Component Concern Overload 3; Connector Envy 3; Scattered Parasitic Functionality 3; Extraneous Connector 2; Extraneous Adjacent Connector 1.
Khomh et al. [311] (2009): Low Cohesion Only 2; Class Global Variable 1; Child Class 1; Class One Method 1; Field Private 1; Has Children 1; Many Attributes 1; Method No Parameter 1; Multiple Interface 1; No Inheritance 1; No Polymorphism 1; Not Abstract 1; Not Complex 1; One Child Class 1; Parent Class Provides Protected 1; Rare Overriding 1; Two Inheritance 1.
Macia Bertran et al. [294] (2011): God Pointcut 3; Redundant Pointcut 3; Composition Bloat 2; Forced Join Point 2; God Aspect 2; Idle Pointcut 2.
Liu et al. [296] (2012): Useless Class 1; Useless Field 1; Useless Method 1.
Yamashita and Moonen [313] (2012): Simulation of multiple inheritance 1.
Arnaoudova et al. [398] (2013): Linguistic Antipatterns 2.
Fard and Mesbah [355] (2013): Empty Catch Blocks 1.
Linares-Vásquez et al. [314] (2014): BaseClassKnowsDerivedClass 1; BaseClassShouldBeAbstract 1; ManyFieldAttributesButNotComplex 1.
Oizumi et al. [320] (2014): Unused Interface 1.

The number at the end of each smell denotes the number of papers that study this smell.

Fowler and Beck [9] describe 22 bad smells; however, the analysis conducted in the references reveals that other bad smells are also credited to Fowler and Beck. This happens because, there is an online catalog that extends the book, and beside the bad smells, Fowler and Beck also present some development conventions considered to be good practice, and the violation of these practices are considered as bad smells by some studies. For example, CLASS DATA SHOULD BE PRIVATE (CDSBP) is a bad smell defined as: *"a class exposing its attributes"* [326], also known as FIELD PUBLIC [311]. Tufano et al. [360] credit this bad smell to Fowler and Beck. However, Fowler and Beck do not explicitly declare this bad smell, but present the concept that the public fields should be converted to private fields and advisor methods should be created (*Encapsulate Field* [9]). Therefore, Fowler and Beck implicitly declare this "bad smell" and hence some researchers credit its origin to them. The same happens with other bad smells from Table 6 (e.g., ANTISINGLETON [319], MISPLACED CLASS [345]).

We also found that the combination of bad smells is a strategy to justify the description of new bad smells. For example, the smell "DUPLICATE CODE IN CONDITIONAL BRANCHES", defined as: *"same or similar code structure repeated within the branches of a conditional statement"* [322], is the combination of SWITCH STATEMENTS [9] and DUPLICATE CODE [9]. The statement *"The problem with switch statements is essentially that of duplication"* [9] confirms our observation.

The interpretation of bad smell subjective definitions also has led to new bad smell descriptions. This can be found in Mäntylä et al. [371], where the authors show that LARGE CLASS can be interpreted using two points of view (Size — LARGE CLASS ONLY [311] and Complexity — COMPLEX CLASS [317]). The same happens with other bad smells (e.g., MANY ATTRIBUTES [311], BRAIN CLASS [324], CLASS ONE METHOD [335]).

The literature also describes original bad smells, i.e., those that appear from the systematic observation or necessity. One example of original bad smells was proposed by Abebe et al. [43], who define a "LEXICON BAD SMELL" as: *"a concept similar to that of a "code smell" and it refers to potential lexicon construction problems, which could be solved by means of refactoring (typically renaming) actions"*. Similarly, other bad smells are also considered original (e.g., COMPOSITION BLOAT [294], FORCED JOIN POINT [294], LINGUISTIC ANTIPATTERNS [398]).

We also found a set of bad smells that only exist in a *specific context*. In terms of *context*, we can identify three categories:

Architectural/design pattern: the smells are proposed/defined based on the corresponding architecture/pattern. Aniche et al. [409] observe that the definition of classical smells does not consider the architecture of the system, limiting their use (e.g., *"In MVC, Data Access Object (DAO) classes are responsible for dealing with the communication towards the databases. These classes, besides not containing complex and long methods (traditional smells) should also limit the complexity of SQL queries residing in them"* [409]). Thus, Aniche et al. [409] provide a catalogue of six smells that are specific to web systems that rely on the MVC pattern. Similarly, Arcelli Fontana et al. [45] shows that some false positive smells can be related to design patterns (e.g., false positives of FEATURE ENVY are related to visitor design pattern).

Environmental: In this category, the runtime environment is considered to propose the catalogue of smells. Hecht et al. [416], proposes an Android bad smell (IGS - Internal Getter/Setter) that occurs when a field is accessed, within the declaring class, through a getter/setter. This indirect access to the field may decrease the performance. This strategy is a common practice in languages like Java because compilers or virtual machines can *inline* the access. However, in Android, the usage of a trivial getter/setter is often converted into a virtual method call, which makes the operation at least three times slower than a direct access. Other examples can be found in [368, 411, 416].

Domain Specific Language (DSL): we found some smells that occur in specific languages, especially in DSLs [46] and do not generalize for general-purpose languages. (e.g., CSS [410], JavaScript [412], Puppet [415]).

The traditional bad smells (Fowler and Beck [9] and Brown et al. [10]) are those that have received more attention

from researchers, although there exists an important body of knowledge enhancing the original catalogs.

Lessons About RQ1.2

We found that researchers have been suggesting new interpretations for the definitions of existing bad smells or even given synonyms for existing bad smells. This is an indication of fragmentation of definitions due to the lack of systematic or/and formal taxonomies for code smells.

6 Results on Interest on Smells Over Time (TA2: when)

This area contains questions on how research on bad smells has attracted interest over time.

6.1 RQ2.1: Has the interest in bad smells evolved over the years?

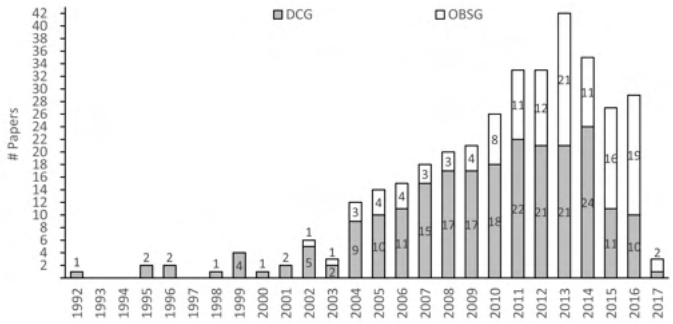

Fig. 2. Distribution of number of papers over time, organized by groups.

Fig. 2 shows the evolution over the years of the number of papers studying Duplicate Code only (DCG) and the other smells (OBSG), respectively. Work on Duplicate Code started 10 years earlier (even earlier than work by Fowler and Beck [9] and work by Brown et al. [10]). In the *Final Database* Duplicate Code is the oldest bad smell. This topic also experienced a higher rate of growth in the number of papers earlier (around 2004), while the other bad smells started receiving more attention only since 2010. This might in part explain why Duplicate Code tend to be studied mainly alone and why the number of papers on this type of smell is much larger than other smell types. These findings again suggest that Duplicate Code can be considered as a topic on its own.

To complement the analysis of Fig. 2, we considered the slope of the regression line for the number of papers in a 20-year period. We considered two groups: 1) all papers in the selected venues, and 2) only papers on bad smells selected for this review. In order to compare the two slopes, the number of papers were normalized by the maximum values of each group, respectively. We found that the slope β for all papers is 0.033 ($Adj.R^2 = 0.906$), where β=0.039 after 2004, and β=0.031 before 2005, i.e., the growth rate is almost the same in both decades. On the other hand, we found that the slope β for papers on smells only is 0.052 ($Adj.R^2 = 0.890$), where β=0.072 after 2004, and β=0.018 before 2005, indicating that after 2004 the bad smells experienced a much higher growth rate than before and higher than other areas on average.

Analytically, between 1992 and 2003, the *Final Database* shows only 22 papers and most of them (90.9%) were dedicated to the study of Duplicate Code (see Fig. 2). In 2004, the number of publications significantly increased. In this year, the Duplicate Code issue has matured because of the steady number of papers after that year, as shown in Fig. 2. We also observe that in this year the books by Brown et al. [10] and Fowler and Beck [9], published in 1998 and 1999, respectively, started to have an impact on the scientific community.

Until the end of 2010, most of the papers (79.0%) studied only Duplicate Code. Between 2011 and 2012, the number of papers on DCG and OBSG remained practically stable (see Fig. 2). The year 2013 presented the largest number of publications on bad smells. Also, in this year the number of papers in the DCG and OBSG groups was equivalent. In 2014, the total number of papers decreased. This reduction was more significant for the OBSG papers. We also observed a decreasing trend of the interest in Duplicate Code in the last years, with a number of papers lower with respect to the other types of smells (e.g., Large Class, Long Method), whose interest still remains high after 2014.

Considering only OBSG papers, Table 7 shows that the interest in the bad smells studied the most (in more than seven papers) is proportionally distributed over the past decades. While these bad smells represent 21.1%, 22 out of a total of 104 bad smells, they are studied in 82.2%, 102 out of 124 papers of the OBSG group. This means that this list is quite representative of the total number of OBSG papers, but more than 78.9% of the different kinds of studied bad smells are not covered in this list, suggesting that for those bad smells more studies may still be required. Until the end of 2003, we did not find publications on the most recurring smells. Thus, these years are omitted from Table 7.

As discussed in RQ1.1, Large Class is the bad smell studied the most in the OBSG papers (see Table 4). Our dataset shows that the first papers about this bad smell date back to 2004 (see Table 7). Until 2007, the interest in this smell remained practically stable. Between 2008 and 2013, the interest marked a sensible growth. The promising results of earlier studies may have possibly boosted the interest since 2008. We also observed that the interest in Large Class had a strong growth between 2010 and 2013 and a strong decline in 2014.

This observation is also valid for the bad smell Long Method, because this smell heavily co-occurs with Large Class (see Table 5). For Feature Envy, the interest grows until 2013 and remains stable for the next two years.

In Table 7, we also observe that the interest in some bad smells (e.g., Shotgun Surgery, Refused Bequest, Long Parameter List, ...) is not continuous over the years and is restricted to small ranges (e.g., 2006 — 2011). Considering only full years (exception of 2017), we can also observe that the last five years are the most representative in terms of variety of bad smells and quantity of studies.

TABLE 7
OBSG papers distributed over time and bad smells.

Bad Smells \ Years	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	Total
Large Class	3 (3.4%)	3 (3.4%)	4 (4.6%)	1 (1.1%)	3 (3.4%)	3 (3.4%)	6 (6.9%)	7 (8.0%)	10 (11.5%)	16 (18.4%)	8 (9.2%)	11 (12.6%)	12 (13.8%)	87 (70%)
Feature Envy	1 (2.0%)		3 (6.1%)	1 (2.0%)	1 (2.0%)	1 (2.0%)	4 (8.2%)	3 (6.1%)	5 (10.2%)	8 (16.3%)	8 (16.3%)	8 (16.3%)	6 (12.2%)	49 (40%)
Long Method	1 (2.1%)	1 (2.1%)	2 (4.2%)	1 (2.1%)	1 (2.1%)	1 (2.1%)	3 (6.3%)	1 (2.1%)	7 (14.6%)	11 (22.9%)	6 (12.5%)	5 (10.4%)	8 (16.7%)	48 (39%)
Data Class	2 (5.4%)	1 (2.7%)	2 (5.4%)	1 (2.7%)	1 (2.7%)	1 (2.7%)		2 (5.4%)	2 (5.4%)	7 (18.9%)	5 (13.5%)	6 (16.2%)	7 (18.9%)	37 (30%)
Shotgun Surgery	1 (3.1%)		2 (6.3%)	1 (3.1%)			1 (3.1%)	1 (3.1%)	3 (9.4%)	7 (21.9%)	4 (12.5%)	8 (25.0%)	4 (12.5%)	32 (26%)
Refused Bequest	1 (3.3%)		2 (6.7%)	1 (3.3%)	1 (3.3%)		1 (3.3%)		4 (13.3%)	8 (26.7%)	4 (13.3%)	4 (13.3%)	4 (13.3%)	30 (24%)
Long Parameter List	1 (3.8%)		1 (3.8%)		1 (3.8%)	1 (3.8%)	1 (3.8%)		7 (26.9%)	6 (23.1%)	4 (15.4%)	1 (3.8%)	3 (11.5%)	26 (21%)
Spaghetti Code					1 (4.3%)		2 (8.7%)	3 (13.0%)	3 (13.0%)	5 (21.7%)	4 (17.4%)	3 (13.0%)	2 (8.7%)	23 (19%)
Message Chains Class			1 (5.6%)		1 (5.6%)	1 (5.6%)	1 (5.6%)		3 (16.7%)	3 (16.7%)	2 (11.1%)	4 (22.2%)	2 (11.1%)	18 (15%)
Lazy Class		1 (5.9%)	1 (5.9%)			1 (5.9%)	1 (5.9%)		4 (23.5%)	2 (11.8%)	4 (23.5%)	1 (5.9%)	2 (11.8%)	17 (14%)
Speculative Generality			1 (0.0%)			1 (6.3%)	1 (12.5%)		3 (6.3%)	4 (12.5%)	3 (18.8%)	1 (12.5%)	2 (12.5%)	16 (13%)
Func. Decomposition					1 (0.0%)	1 (6.3%)	2 (6.3%)	3 (0.0%)	1 (18.8%)	2 (25.0%)	3 (18.8%)	2 (6.3%)	1 (6.3%)	16 (13%)
Data Clump			1 (6.7%)		1 (6.7%)		1 (6.7%)		2 (13.3%)	4 (26.7%)	2 (13.3%)	3 (20.0%)	1 (6.7%)	15 (12%)
Swiss Army Knife					1 (7.1%)		2 (14.3%)	1 (7.1%)	3 (21.4%)	3 (21.4%)	1 (7.1%)	1 (7.1%)	2 (14.3%)	14 (11%)
Complex Class Only						1 (7.1%)			2 (14.3%)	3 (21.4%)	2 (14.3%)	3 (21.4%)	3 (21.4%)	14 (11%)
Field Public (CDSBP)						1 (0.0%)			2 (7.7%)	4 (15.4%)	2 (7.7%)	2 (0.0%)	2 (15.4%)	13 (10%)
Misplaced Class	1 (8.3%)								3 (25.0%)	5 (41.7%)	1 (8.3%)		2 (16.7%)	12 (10%)
Divergent Change			1 (0.0%)				1 (0.0%)		3 (16.7%)	1 (33.3%)	1 (16.7%)	4 (16.7%)	1 (8.3%)	12 (10%)
Temporary Variable		1 (0.0%)	1 (0.0%)						1 (0.0%)	4 (22.2%)	1 (11.1%)		1 (11.1%)	9 (7%)
Brain Method							1 (0.0%)		1 (11.1%)	1 (44.4%)	1 (11.1%)	3 (0.0%)	2 (22.2%)	9 (7%)
Intensive Coupling							1 (12.5%)		1 (12.5%)	2 (25.0%)	1 (12.5%)	2 (25.0%)	1 (12.5%)	8 (6%)
Extensive Coupling							1 (12.5%)			2 (12.5%)	1 (12.5%)	2 (50.0%)	2 (25.0%)	8 (6%)

Lessons About RQ2.1

The number of papers on DUPLICATE CODE had increased from 2004 to 2014. From 1992 to 2003 few studies were conducted and an important decrease can be observed since 2015.

The first paper concerned with other types of smells dates back 2002 and papers about these smells experienced a consistent increase after 2010, registering a peak in 2013.

We observe that most of the earliest smells in the OBSG group (e.g., Brown et al. [10]) have not received much attention, and there seems to be no trend toward change.

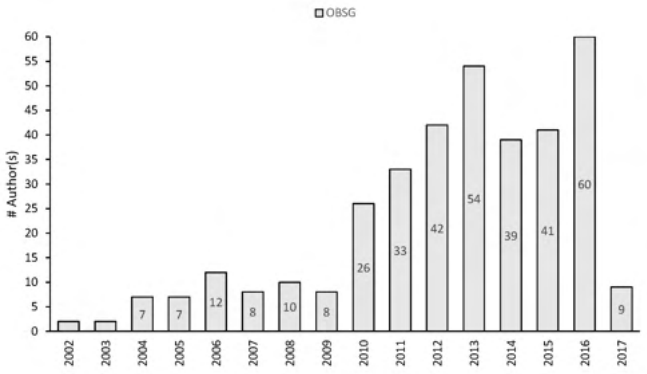

Fig. 3. Absolute number of authors over time (OBSG papers).

6.2 RQ2.2: Has the research community interested in bad smells evolved over the years?

RQ2.1 reported on the number of papers over the years for the different kinds of bad smells. This question complements RQ2.1 from the point of view of number of authors. However, in this research question we only consider OBSG papers because this is our main focus.

Fig. 3 presents the distribution of the number of authors that published papers on bad smells, between 2002 and 2017. In this case, all papers in the OBSG group are considered. An author is counted once for each year he/she has published a paper.

Moreover, Fig. 4 shows the cumulative number of authors between 2002 and April 2017. Here, each author is counted only once in the year of his/her oldest paper and each year adds to the cumulative number of authors of the previous year the number of newcomer authors publishing that year.

In our systematic literature review, we collected the papers published between the years 1990 and 2017. However, until the end of the year of 2001, we did not find publications in the OBSG group. Thus, these years are omitted from Fig. 3 and Fig. 4.

Until the end of 2009, the absolute number of authors was relatively low and remained virtually constant (see Fig. 3). On the other hand, the cumulative number of authors had grown moderately and, in this period, there were only 48 distinct researchers publishing papers about bad smells (see Fig. 4).

Between 2010 and 2013, the number (absolute and cumulative) of authors significantly increased (see Fig. 3 and 4). We observe that in these years, the studies by Brown et al. [10] and Fowler and Beck [9], published respectively in 1998 and 1999, started to gain more visibility. Thus, a new set of smells started to be studied (see Table 7).

In 2014 and 2015, the absolute number of authors de-

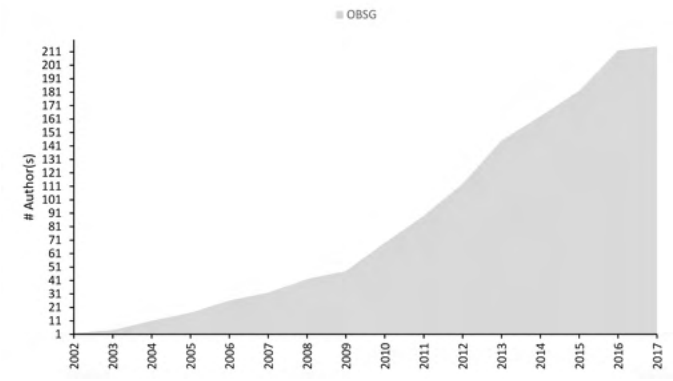

Fig. 4. Cumulative number of distinct authors over time (OBSG papers).

creased (see Fig. 3). However, the cumulative number of authors keeps growing steadily (see Fig. 4) which means that in this period, new authors are still being attracted by this research topic. This is confirmed the following year (2016), where both figures show an increase on the absolute and cumulative numbers. Interestingly, in 2016 the number of different authors is the highest one, but the number of papers did not increase proportionally, indicating that there were more authors per paper in this year, evidencing an increase in the mean size of groups.

Lessons About RQ2.2

Bad smell is a growing research topic that has been continuously attracting interest by new researchers exploring several different perspectives. Generally, we observe that this research topic became quite popular and it is gaining more visibility in the scientific community.

7 Results on Aims, Findings and Settings (TA3: what)

This section presents answers to questions related to the aims (main goals of papers) and the experimental setting (tools and subject systems used to evaluate the research). As explained earlier, Duplicate Code is not considered. We only consider papers where Duplicate Code is studied together with other smells.

7.1 RQ3.1: Which are the most commonly targeted aims?

In order to answer this research question, we manually identified the purposes of studies. These aims were not predefined, but they emerged from the qualitative analysis of papers, where we coded the sentences containing the aims of the papers. We describe the thirteen different purposes identified:

1) Detection: the goal is to detect bad smells in source code. Generally, these papers also compare the new detection technique with existing approaches.
2) Inter-relationship: the goal is to show how bad smells are related to each other.

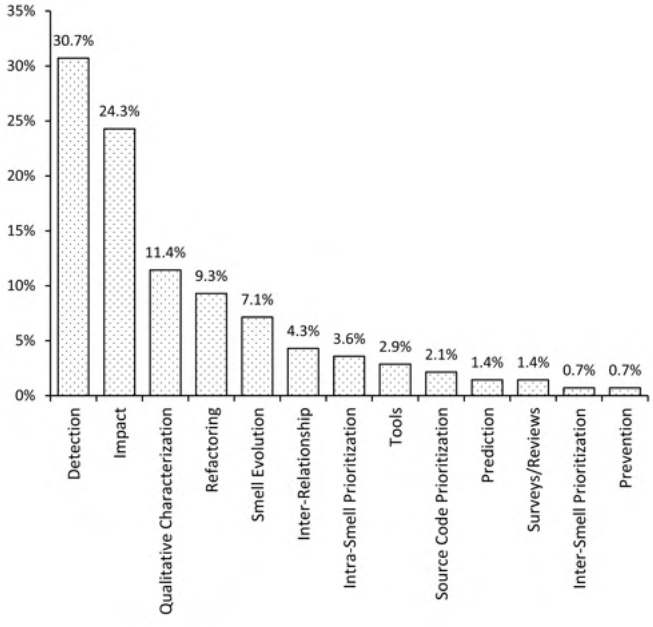

Fig. 5. Aims of OBSG papers.

3) Qualitative characterization: the goal is to comprehend the underlying characteristics of bad smells. Generally, this kind of work tries to map the subjective human perception of bad smells to more objective "rules/standards", sometimes proposing new types of bad smells.
4) Refactoring: the goal is to apply and/or create techniques to remove/correct bad smells. Generally, these studies involve a detection step, as secondary goal.
5) Impact: the goal is to understand how bad smells impact on attributes related to maintenance, quality and/or evolution of software. Sometimes, they employ refactoring techniques to quantify the impact of bad smells on attributes related to maintenance.
6) Prevention: the goal is to build and/or apply techniques to prevent the introduction of bad smells.
7) Smell evolution: the goal is to understand how bad smells evolve over time and how factors related to software maintenance evolve accordingly. Generally, historical smell evolution is based on the analysis and comparison of several versions of source code from the project repository. We considered in this category descriptive studies. If the aim is to predict the future, we classify the paper in the following distinct category.
8) Prediction: the goal is to predict the future behavior of factors related to maintenance, e.g., predict software maintainability based on bad smell information.
9) Surveys/reviews: the goal is to compile a summary of the main facts and findings related to bad smells. The compilation can be made based on questionnaires answered by professionals or even checking the scientific literature.
10) Tools: the goal is to propose tools/techniques to document and visualize the types of bad smells.
11) Intra-smell prioritization: given a set of code entities with the same type of bad smell (e.g., Data Class), the

goal is to identify the most appropriate order of entities to analyze/refactor.

12) Inter-smell prioritization: given a set of different types of bad smells (e.g., SHOTGUN SURGERY, FEATURE ENVY), the goal is to investigate which set should be the subject of structuring/analysis. Therefore, given a set of different types of smells, they identify which type of bad smell should be refactored first.
13) Source code prioritization: it is a mix between Intra- and Inter-smell prioritization. In other words, given a set of different types of bad smells and a subset of infected code, for each kind of bad smell, the goal is to identify which code elements should be subject to maintenance.

Fig. 5 proportionally presents the aims found in the OBSG papers. In the first position (30.7%), we have papers proposing tools and techniques intended to **detect** different types of bad smells. This category accounts for every paper that proposes a tool/technique to identify bad smells, regardless of the strategy (e.g., Metrics-based [339], Parallel Evolutionary Algorithms [340], SVM-based [364], Bayesian approach [373, 408]), or the development environment. Most *Detection* papers (96.7%) are focused on detecting bad smells based on the analysis of the source code.

One remark is that the *Detection* aim is highly related to the *Impact* aim, because papers that aim at analyzing the impact of bad smells, typically, have first to detect them. Indeed, we observe that the inconclusive knowledge about the negative impact of bad smells can be partially attributed to tools/techniques that detect them, explaining the large number of studies also found in these categories. Due to the fact that there is a high variety of tools, and discrepancies on what those tools find, discrepant results in different studies using different tools cannot be discarded; as already observed by Pate et al. [6]: *"there are contradictions among the reported findings"*. Moreover, the inherent difficulties to have bad smell detectors with high precision and recall may also explain part of the large interest.

In the second position (24.3%), we found studies focused on verifying how the occurrence of bad smells **impacts** several factors (e.g., Quality [299], Change-proneness [311], Fault-proneness [331], Defect-proneness [367]) related to software maintenance/evolution. This may be explained by the necessity of answering if tackling bad smells actually brings relevant benefits. Moreover, the number of factors and their possible combinations with distinct smells are high, thus requiring different studies.

The fact that the study of *Impact* has been receiving so much attention until now suggests that there are still no comprehensive and sufficient evidence on the extent of negative effects associated with bad smells on software maintenance and evolution. For instance, Yamashita [324] states that *"Deligiannis et al. (2003) reported that* GOD CLASS *indicated problems, while Abbes et al. (2011) concluded that a* GOD CLASS *in isolation is not harmful"*.

As shown in Fig. 5, *Qualitative Characterization* and *Refactoring* are in third (11.4%) and fourth (9.3%) positions, respectively. *Refactoring* papers are those providing strategies to code refactoring and removal of undesirable effects of bad smells (e.g., [391, 392, 393]). Papers classified as *Qualitative Characterization* are those aimed at comprehending the bad smell mechanisms. They generally transcribe the subjective human perception into objective rules and sometimes propose new bad smells (e.g., [312, 326, 371]).

The other categories are less prevalent, including no more than four papers per category. Moreover, there are aims still lacking studies, for example, smell priorization or prevention. These categories suggests research opportunities. However, to explore these new areas, some challenges must be overcome (e.g., bad smell detectors which have high precision and recall independently of the analyzed project).

In our classification, some papers (19) were classified in more than one category, as they were considered as having multiple aims. The paper by Hall et al. [357] is an example: the authors develop a tool to detect five bad smells (DATA CLUMPS, SWITCH STATEMENTS, SPECULATIVE GENERALITY, MESSAGE CHAINS, and MIDDLE MAN) and also quantify their effects on software faults.

Lessons About RQ3.1

We identified thirteen categories of aims and some of them are more frequent than others. The top-2 aims are *Detection* and *Impact* papers that aim at analyzing the impact of bad smells, typically, have first to detect them. Moreover, the reports on the impact of bad smells sometimes contradict each other. This may explain the large interest on assessing the impact of bad smells. Some aims still lack studies (e.g., *Smell Prevention*). Thus, we identified research opportunities by quantifying the aims.

7.2 RQ3.2: What are the main reported findings?

To extract the main findings from OBSG papers, first we manually read the content of the paper abstract. If not enough, then we examined the conclusions. Furthermore, when necessary, we read the other parts of the paper to find their research questions and respective answers. For all OBSG papers, we have stated their main findings. Next, we analyzed those findings using thematic analysis [47], a technique for identifying and recording patterns (or "themes") within a collection of documents. Thematic analysis involves the following steps: 1) initial reading of the findings, 2) generating initial codes for each finding, 3) searching for themes among codes, 4) reviewing the themes to find opportunities for merging, and 5) defining and naming the final themes. These steps were performed independently by one author of the paper and revised by another author until reaching a consensus.

During the thematic analysis, we observed that certain terms are used inconsistently. According to Tian [48], the terms *defect*, *failure*, *fault*, *error* have specific meaning. However, we concluded that these terms have been used as synonyms (bug [317], fault [331], error [348], defect [378]). Our conclusion relies on the fact that their datasets are based on issue-tracking systems (e.g., Bugzilla), and thus refers to the same entity. In this paper, we will standardize those names using the term *bug*.

We also observed that similar findings were written in different ways, e.g., Olbrich et al. [378] reports that smelly entities contain more defects than other kinds of entities

TABLE 8
Main findings (OBSG papers).

ID	Type	Theme	Convergence	References
01	Maintenance	Adaptive activities are frequently used to remove smells		[376]$^{\circ}$
02	Maintenance	Refactoring contributes to alleviate or remove of smells	✓	[381]$^{\circ}$ [391]* [393]*
03	Maintenance	Refactoring is not related to the quality indicators (e.g., metrics)		[308]$^{\circ}$
04	Maintenance	Refactoring is not frequently used to remove smells	✓	[308]$^{\circ}$ [369]$^{\circ}$ [376]$^{\circ}$
05	Maintenance	Refactoring can introduce smells	✓	[360]$^{\circ}$ [388]*
06	Maintenance	Refactoring can move smells		[387]*
07	Maintenance	Refactoring arbitrary smells does not reduce bug-proneness		[357]$^{\bullet}$
08	Maintenance	Refactoring smelly code does not reduce maintenance effort		[323]$^{\circ}$
09	Maintenance	Low level refactoring (e.g., narrowly-scoped changes) does not remove architecturally-relevant smells		[318]$^{\circ}$
10	Maintenance	Refactoring effort is related to smell refactoring sequence		[296]$^{\bullet}$
11	Maintenance	Smells are frequently ignored in favor of fixing bugs		[321]$^{\circ}$
12	Maintenance	Entities statically related to smelly code are refactoring-prone		[334]$^{\circ}$
13	Change-prone	Entities statically related to smelly and design-patterned entities are change-prone		[332]$^{\circ}$
14	Change-prone	Smelly entities are change-prone	✓	[311]$^{\circ}$ [319]$^{\bullet}$ [325]$^{\circ}$ [378]$^{\circ}$ [380]$^{\circ}$
15	Change-prone	Some smells are more change-prone than other smells	✓	[311]$^{\circ}$ [319]$^{\bullet}$ [325]$^{\circ}$
16	Change-prone	Smelly design model entities are change-prone		[370]$^{\circ}$
17	Bug-prone	Entities having co-changing dependencies with smelly entities are bug-prone		[331]$^{\circ}$
18	Bug-prone	Smelly design models entities are bug-prone		[370]$^{\circ}$
19	Bug-prone	Smelly entities are bug-prone	✓	[295]$^{\circ}$ [317]$^{\circ}$ [319]$^{\bullet}$ [334]$^{\circ}$ [348]$^{\circ}$ [357]$^{\bullet}$ [378]$^{\circ}$ [412]*
20	Bug-prone	Entities participating in static dependency or co-change with smelly entities are bug-prone	✓	[331]$^{\circ}$ [334]$^{\circ}$ [367]$^{\circ}$
21	Positive Aspect	Some instances of smells are the best solution		[387]*
22	Positive Aspect	Some kinds of smells or some smell instances are less bug-prone	✓	[295]$^{\circ}$ [357]$^{\bullet}$
23	Positive Aspect	Entities participating in static relationships between smells and design patterns are less bug-prone than other smell classes		[332]$^{\circ}$
24	Association	Automatically-detected smells are not correlated with architectural problems		[315]$^{\bullet}$
25	Association	Smelly entities are associated with architectural problems		[318]$^{\circ}$
26	Association	Smells are associated with the increment of maintenance effort (e.g., editing, navigating)	✓	[324]$^{\bullet}$ [330]$^{\circ}$
27	Association	Some smells are associated with specific design pattern		[332]$^{\circ}$
28	Association	Some smells are associated with test smells		[351]$^{\bullet}$
29	Association	Some smells are correlated with metrics	✓	[299]$^{\circ}$ [312]$^{\circ}$ [314]$^{\circ}$ [371]$^{\circ}$
30	Association	Some smells are not correlated with metrics	✓	[312]$^{\circ}$ [371]$^{\circ}$
31	Association	The number of smells impact on the values of metrics		[299]$^{\circ}$
32	Association	Smells have a negative impact on the developers' performance (e.g., time)	✓	[304]$^{\circ}$ [382]$^{\bullet}$
33	Association	The influence of smells on maintainability is relatively small	✓	[324]$^{\bullet}$ [328]$^{\bullet}$
34	Association	The intensity of smells may or may not represent a problem		[326]$^{\circ}$
35	Association	The kind of smell impacts on the types of changes		[325]$^{\circ}$
36	Association	Some smells could be domain-dependent	✓	[299]$^{\circ}$ [314]$^{\circ}$ [383]$^{\circ}$
37	Association	Some maintainability factors (e.g., design consistency) have relation to the definitions of smells		[313]$^{\circ}$
38	Association	Smells have a negative impact on the energy consumption		[411]$^{\circ}$
39	Association	Smells have a negative impact on the memory performance		[416]$^{\circ}$
40	Association	File size impacts reading and searching activities more than smells		[330]$^{\circ}$
41	Perception	Smelly entities perception is influenced by professional position (e.g., lead developer)		[312]$^{\circ}$
42	Perception	Smelly entities are perceived as poor practices		[401]*
43	Perception	Some smells are not perceived as design problems		[326]$^{\circ}$
44	Perception	Some smells are easily identified		[400]*
45	Detection Techniques	New strategies to identify smells (e.g., based on Machine Learning)	⧄	[300]$^{\circ}$ [301]$^{\circ}$ [303]$^{\circ}$ [309]$^{\circ}$ [329]$^{\bullet}$ [345]$^{\circ}$ [354]$^{\circ}$ [356]$^{\circ}$ [359]$^{\circ}$ [361]$^{\circ}$ [375]$^{\bullet}$ [383]$^{\circ}$ [385]$^{\circ}$ [396]* [397]*
46	Detection Techniques	Proposed approach is more efficient/effective than state of the art	⧄	[310]$^{\circ}$ [335]$^{\bullet}$ [337]$^{\circ}$ [340]$^{\circ}$ [347]$^{\circ}$ [358]$^{\circ}$ [364]$^{\circ}$ [365]$^{\bullet}$ [366]$^{\bullet}$ [373]$^{\circ}$ [391]* [392]* [393]* [408]*
47	Detection Techniques	Different detectors for a same smell produce different answers		[350]$^{\circ}$
48	Detection Techniques	Bug prediction based on smells are improved including new factor(s) (e.g., history information of smells)	✓	[317]$^{\circ}$ [352]$^{\circ}$ [395]*
49	Life Cycle	The number of smelly components increases/decreases in the periods of time		[380]$^{\circ}$
50	Life Cycle	Technical debt (e.g., code debt, documentation debt) increases over time due to the introduction of new instances that are not fixed		[298]$^{\circ}$
51	Life Cycle	Smells are born with the entity	✓	[351]$^{\bullet}$ [360]$^{\circ}$ [369]$^{\circ}$ [376]$^{\circ}$
52	Life Cycle	Entities with low review coverage or overloaded developers are smell-prone	✓	[344]$^{\circ}$ [343]$^{\circ}$ [360]$^{\circ}$
53	Relationship	Interaction of smells (e.g., collocated/coupled) are associated with maintenance problems	✓	[324]$^{\bullet}$ [327]$^{\bullet}$ [328]$^{\bullet}$
54	Relationship	Agglomerations of smells and certain topologies are more related to architectural problems	✓	[320]$^{\bullet}$ [349]$^{\bullet}$
55	Relationship	Agglomerations of smells and certain topologies are more related to design problems		[346]$^{\bullet}$
56	Catalog	New set of smells is proposed	✓	[301]$^{\circ}$ [354]$^{\circ}$ [361]$^{\circ}$ [385]$^{\circ}$

*Single occurence of smell; $^{\circ}$Co-occurrence of smells; $^{\bullet}$Co-study of smells; ✓Convergent finding; ⧄A special type of convergent finding.

and Li and Shatnawi [348] reports that some smells were positively associated with the probability of error in the respective entities. In these situations, we standardized the sentences in order to merge the findings into a theme (e.g., smelly entities are bug-prone).

Table 8 shows the themes extracted from the findings in OBSG papers. This table also shows the consensus/convergence of findings. Each finding reported by two or more papers is considered as converging. Following our definition of convergence, we observed some reasons for that convergence:

- i) extension paper: an extended version of a particular paper is published in other venue. In general, this occurs because authors submit their preliminary results to the conferences and afterwards, with the consolidation of research, a new paper is published in a journal and this paper contains the old and the new findings (e.g., [369, 376]).
- ii) replication paper: the study presents a replication of previous work. For this case, in general, the replication paper is conducted by other authors and the experimental setup could have variations, but the goal and the analysis process are similar, e.g., the paper [298] presents a replication of the work by Potdar and Shihab [49]. The paper [329] replicate the findings from previous work ([327]) on inter-smell relations by analyzing

larger systems and by including both industrial and open source ones.
- iii) coincidence: Two or more independent studies conclude similar things. This situation is similar to the replication paper, but the papers do not have similarities on the analysis method, for example: Chatzigeorgiou and Manakos [369] investigated the frequency with which the refactoring activities are applied on smelly code, Bavota et al. [308] investigated which are the refactoring activities applied on smelly components. Both papers conclude that the refactoring activities are not frequently used to remove smells.

The convergence of a finding does not necessarily means that it can be generalized to any smell/system/situation, e.g., the finding "smells are born with the entity" [351, 360, 369, 376] cannot be generalized to LEXICON BAD SMELLS because we did not find empirical studies that analyze when such smells are introduced. Instead, the convergence of a finding helps researchers to explore in details the papers that are similar. Regarding the findings classified as *Detection Techniques*, we also consider that they are convergence findings (☑) because we observed a "consensus" on their aims and results (e.g., investigate strategies to identify smells and somehow improve other baselines). The data reveals that 40% of the findings are convergent (see Table 8).

Table 9 shows the main contradictions/divergences among the findings. The findings listed in the second and third column of Table 9 are the same findings as in Table 8 organized to make explicit the divergences between the findings that are on the same row. For example, in the first row (ID 01), we have a conflict on findings that details how the smells are removed. In this case, considering the numerous catalogs [9, 10] and tools designed to help the developers on smell refactoring (e.g., JDeodorant [50]), it looks like refactoring is the principal treatment to remove smells but the findings listed on this columns are opposite to this observation (e.g., refactoring is not frequently used to remove smells). In other words, these two columns detail the findings that have some level of divergence (e.g., a paper claimed the finding "X" and another paper reports an opposite result). Table 9 also has a column *implications and challenges* that describes the consequence of the divergences, e.g., standard refactoring catalogs could be rethought. This column also details the issues that must be addressed or considered in further studies. There are several levels of divergences and this depends on human perception, thus our strategy follows these steps: two authors examined independently each finding listed in Table 8 and using their expertise produced a list of findings that have some level of divergence. Next, these lists are revised by these authors until reaching a consensus. In the end, we found seven divergences and they are detailed on Table 9. In general, the conflicts and their implications are related to the limitations and threats to validity of papers. Challenges are associated with experimental setup/environment. Some findings of Table 8 (e.g., ID 56) are not listed on Table 9, because they do not have a divergence with other findings in Table 8.

In the next paragraphs, we report the some observations with respect to the findings and/or divergences from papers.

From the perspective of *Maintenance*, we observed surprising findings, e.g., refactoring is not frequently used to remove smells. Instead, they are removed by adaptive activities, suggesting that adaptive activities could be analyzed and used to improve refactoring catalogs. Nonetheless, entities statically related to smelly code (ANTISINGLETON, LONG METHOD, COMPLEX CLASS, LONG PARAMETER LIST, MESSAGE CHAINS, SWISS ARMY KNIFE, REFUSED BEQUEST, SPECULATIVE GENERALITY) are refactoring-prone [334], meaning that smelly entities may be useful to drive refactoring but not necessarily on themselves.

We also observed that some smells are removed with multiple refactoring operations. A LARGE CLASS has many non-inter-related responsibilities, suggesting that part of its behavior could be split into other components. Thus, *Extract Class* and/or *Extract Subclass* may be performed several times, but developers may not necessarily refactor all unrelated responsibilities. Considering this observation, the question if refactoring removes or not smells was answered with an indication that refactoring is not frequently used to remove smells [308, 369, 376]. However, we can argue that refactoring operations are used to reduce the symptom of bad smells or even their intensity. This observation can be more prevalent in maintenance tasks where the time and/or the complexity are critical factors. This view may in part explain why smelly code are refactoring-prone, but refactoring operations do not remove the smells.

We would have an indication that refactoring operations are useful to improve the maintainability of projects, because there is some evidence that the smells are associated with maintenance effort [324, 330], and because there are multiple guidelines to remove smells (e.g., [9, 10]). However, some empirical studies show that refactoring smelly code does not reduce maintenance effort [323]. Thus, these findings are contradictory (see the second item in Table 9).

The main findings also reveal some unusual things. Some instances of smells are the best solution in specific contexts. This finding is conflicting because, in general, smelly entities are perceived as poor practices (see the last item in Table 9). According to Vaucher et al. [387], LARGE (GOD) CLASSES are sometimes embodied in design as the best solution to a particular problem. Although they are not "good" code, these classes cannot be improved and remain relatively untouched from version to version. However, considering that recently developed projects, in principle, could be more likely to follow good modern practices than old projects, the reproduction of this study could produce a different result with newer projects.

Regarding *Change/Bug Proneness* and *Positive Aspects*, there is an interesting relationship. An unexpected finding is that some kinds of smells or some smell instances are less bug-prone. Hall et al. [357] report that MESSAGE CHAINS which occurred in larger files reduced bugs, being valid for all analyzed systems. They also report that DATA CLUMPS reduce bugs, but only in two systems. Similar results are reported for MIDDLE MAN, SPECULATIVE GENERALITY, SHOTGUN SURGERY and FEATURE ENVY [295, 357].

These "positive aspects" contradict the "harmful aspects" of smells. A lot of papers report that smelly entities are bug-prone [295, 317, 319, 334, 348, 357, 378, 412]. In this context, the smell MESSAGE CHAIN is associated with bugs

TABLE 9
Main findings and their divergences (OBSG papers).

ID	Main Findings	Conflicting Findings	Implications and Challenges
01	✸Refactoring contributes to alleviate or remove of smells [9, 51, 381, 391, 393].	✸Refactoring is not frequently used to remove smells [308, 369, 376]. ✸Adaptive activities are frequently used to remove smells [376]. ✸Refactoring can introduce smells [360, 388]. ✸Refactoring can move[1] smells [387].	These conflicting findings could suggest developers to disregard refactoring on smelly entities. Considering that empirical studies have shown that refactoring takes places within adaptive activities, deciding on adequate refactoring suitable for the specific evolution context is still a challenge.
02	✸Smells are associated with the increment of maintenance effort (e.g., editing, navigating) [324, 330].	✸Refactoring smelly code does not reduce maintenance effort [323].	The developers' could produce non standard strategies (e.g., specific for a project) to control smelly codes. In other words, standard refactoring catalogs could be rethought. There are many factors associated with the maintenance effort. Thus, we need studies to explore in detail the different factors related to maintenance effort and their relationship with code smells and refactoring.
03	✸Smelly entities are bug-prone [295, 317, 319, 334, 348, 357, 378, 412].	✸Some kinds of smells or some smell instances are less bug-prone [295, 357]. ✸Entities participating in static relationships between smells and design patterns are less bug-prone than other smell classes [332].	Strategies of bug-prediction/control based on smells are less effective than it could be. This reveals the need to map the situations in which the smell instances are bug-prone and from those that are less bug-prone. The discovery of patterns, enable us to distinguish the smell instances that are bug-prone from those that are less bug-prone. This could improve the control of bugs based on smells.
04	✸Some smells are correlated with source code metrics [299, 312, 314, 371].	✸Some smells are not correlated with source code metrics [312, 371].	This implies that the detection strategies based exclusively on metrics are limited and this causes distrust among the users and the fear might prevent adoption of smell detection tools in practice. The challenge is related to defining effective detection strategies to complement the ones based on metrics.
05	✸Smelly entities are associated with architectural problems [318]. ✸Agglomerations of smells and certain topologies are more related to architectural problems [320, 349].	✸Automatically-detected smells are not correlated with architectural problems [315].	Aspects related to architectural issues are neglected by tools that automatically detect the smells. This indicates the necessity of other studies that investigate the association/correlation between the architectural problems and the smelly entities. The set of smells and architectural problems that are relevant and representative is important in the future works.
06	✸Smells have a negative impact on the developers' performance (e.g., time) [304, 382].	✸The influence of smells on maintainability is relatively small [324, 328].	These findings suggest that developers' performance could be an aspect separated from the concept of maintainability of systems. However, the developers' performance is intrinsically linked to maintainability. Thus, we have some level of contradiction in the different findings and we suggest new studies devoted to cover issues related to the software maintainability and their relationship with smells.
07	✸Smelly entities are perceived as problems and/or poor practices [401].	✸The intensity of smells may or may not represent a problem [326]. ✸Some instances of smells are the best solution [387].	Even the human perception does not reflect all cases observed in the real world. Thus, there is the necessity of future studies to map and evaluate the subjective human perception. In particular, using smelly codes classified as poor practices and those classified as the best solution.

[1]Move smells from one part of the code to another.

[317, 319, 334, 357]. However, these findings are not uniform on all the analyzed systems. In other words, some instances of these smells are bug-prone and others are non-bug-prone. We also observe this situation for other kinds of smell, e.g., SPECULATIVE GENERALITY [317, 319, 334, 357], SHOTGUN SURGERY [295, 348], DATA CLUMP [357]. Thus, the apparent contradiction (see the third item in Table 9) is not strong because there is a factor of context (e.g., human effect, subject system). This suggests that the effect of smells is particularly dependent on the context changing from a system to another.

Regarding to findings of type *Association*, there are many tools for smell detection, and most of them are based on source code metrics. They use this strategy because smelly entity impacts on the values of metrics [299, 314]. However, some smells are not correlated with source code metrics [312, 371]. Thus, we observed that the correlation between metrics and code smells is not a general finding (see the fourth item in Table 9). According to Mäntylä et al. [371], instances of LARGE CLASS are not strongly correlated with metrics based on the number of attributes and/or operations (e.g, LOC). On the other hand, Arcelli Fontana et al. [299] report that LARGE (GOD) CLASS has a strong correlation with the metric Access To Foreign Data (ATFD). Mäntylä and Lassenius [312] report that this and other smells can be difficult to detect because they can be measured in many different ways. These findings show that proposing associations have some traps, and finding cause-effect relations are challenging because there are many factors to be considered.

Considering the main tools (see Subsection 7.4) that automatically detect smells, they have limitations on identifying the interaction of smells (e.g., collocated/coupled). Thus, the finding "automatically-detected smells are not correlated with architectural problems [315]" could be seen as a consequence of their limitations, especially because this finding is divergent with respect to others (agglomerations of smells and certain topologies are more related to architectural problems [320, 349]), as shown by the fifth item in Table 9.

With respect to the theme *Detection Techniques*, as reported in Table 8, we observe that different detectors for the same smell produce different answers (47), which is coherent with the need for new strategies to identify smells (45) or even approaches more efficient/effective than state-of-the-art (46). The number of papers concerned with this theme also suggests some trend: the tools/techniques need to be improved (e.g., precision, recall). In this context, some papers report an improvement in the state-of-art (e.g., [310, 347]). However, as observed in Section 7.4, a lot of them do not share their implementations limiting their applicability (e.g., [306, 307]). Moreover from the perspective of open

science, the unavailability of implementations hinder reproducibility and impose barriers to the underlying empirical studies, in particular for those aiming at comparing new approaches with the state-of-the-art.

Regarding to the theme *Relationship*, the interaction of smells is another investigation trend. In this case, the interaction between smells (e.g., use, collocated, coupled) seems to be more meaningful than the isolated occurrence. Some state-of-the-art tools (e.g., DECOR [366]) corroborate on that using a combination of smells to identify other kinds of smells (e.g., SPAGHETTI CODE is related to LONG METHOD). There are also papers reporting on the association between agglomerations of smells and maintenance/architectural/design problems (e.g., [346, 349]).

Finally, one trend is related to improving the *Catalog* of smells for embracing technological transformations (e.g., mobile device). As shown in Section 5.1, this occurs in two forms: 1) some papers propose specific smells, based on their need (e.g., architecture [409], environment [416]); 2) others use the interpretation of subjective smell definitions to define a new set of smells (e.g., DUPLICATE CODE IN CONDITIONAL BRANCHES [322]). On the other hand, technological transformation in conjunction with deficiencies of techniques/tools lead to challenges. Some papers report that smells have a negative impact on the energy consumption and/or memory performance, especially in mobile devices (e.g., [330, 411]). Thus, app stores could, in principle, use this kind of information to classify the quality of their apps. However, these stores do not provide access to the source code for all apps, imposing a challenge for the current techniques/tools.

Lessons About RQ3.2

Regarding the specific findings, although refactoring has been proposed to remove smells, there are several subtleties that make this activity inherently complex, as shown in the findings *Maintenance*. Although, in general, smelly entities and their dependents are more bug/change-prone, there are some exceptions.

There are several studies that associate (or not) smells with some other factor, however, more in-depth studies on cause-effect are still lacking to provide conclusive evidence. For example, some findings show that smells can be or not correlated with code metrics, showing that other factors, such as human perception, should be considered.

There is a large body of knowledge on techniques to detect smells that aims at improving the state-of-the-art, despite the necessity for more adoption of open science regarding these techniques.

Technical debt tends to increase over the life cycle, although smells typically are existing since the first version of a code component.

7.3 RQ3.3: Considering the co-occurrence of bad smells in the papers of our dataset, how many of them actually study some relations between bad smells and what are the main findings of these co-studies?

RQ1.1 reported on the most common co-occurrences of bad smells in papers. The co-occurrence may be merely coincidental. This question complements RQ1.1 because we investigate not only the frequency of smells being studied in the same paper, but deepen on the aims and results related to the inter-relationship between those smells.

In order to answer this question, we selected the top-5 smells (Table 4), and those papers which showed co-occurrences of smells (Table 5) were examined in order to find the co-studies. Recall that we consider as co-studies those papers where the co-occurrence of smells in the paper is intentionally designed to investigate some inter-relationship or interaction between them.

Our dataset shows 93 papers classified as “co-occurrence only" or “co-study" (Table 15 details this classification to each OBSG paper). Co-occurrence-only papers are the most common (79.5%). Co-study papers are focused on identifying the instances of smells which are relevant according to some criterion, e.g., instances[5] whose refactoring improve the maintainability. For example, Oizumi et al. [320] co-studied smells to identify architectural problems, and observed that agglomerations of smells identify architectural problems significantly better than individual instances. Similarly, Yamashita and Moonen [327] observed that the interactions between bad smells affect maintenance. We also observe that the interactions between smells are obtained by applying techniques that correlate the co-occurrence of smells on the same artifact (e.g., Principal Component Analysis — PCA [327]) and then, the statistically significant correlations are qualitatively analyzed to explain them.

Table 10 is similar to Table 5 in RQ1.1; however, this table focuses **only** on bad smells that were co-studied (e.g., LARGE CLASS was co-studied with DATA CLASS in 8 papers). We observe that only 2 out of the 18 papers where DUPLICATE CODE co-occurs with other smells, are actually co-study papers. Other types of smells also reveal a low number of co-studies with respect to co-occurrences. In the following, we report the main results on co-studies of smells.

DUPLICATE CODE. Although this smell is mostly studied alone, there are still some co-studies of DUPLICATE CODE with other smells. Liu et al. [296] uses the possible relationship between LONG METHOD and DUPLICATE CODE to support prioritization of refactoring because removing DUPLICATE CODE would make the LONG METHOD disappear. According to Fowler and Beck [9], this relationship exists *“When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables,* DUPLICATE CODE *cannot be far behind”*. Similarly, Parnin et al. [336] also report this relation, namely: *“the (long) method is often difficult to understand and may contain* DUPLICATE CODE*”*. However, this assumption is not true for all instances of LONG METHOD and DUPLICATE CODE (e.g., refactoring the clones on a LONG METHOD with

5. Some instances of smells do not affect the maintainability, e.g., DATA CLUMP indicates fewer maintenance problems [324].

TABLE 10
Top five bad smells — map of co-studies of smells.

	Duplicated Code (2\18)		Large Class (15\79)		Feature Envy (9\46)		Long Method (12\47)		Data Class (11\37)	
# Papers	Feature Envy	2	Data Class	8	Large Class	8	Large Class	7	Large Class	8
	Large Class	2	Feature Envy	8	Long Method	6	Feature Envy	6	Feature Envy	5
	Long Method	2	Long Method	7	Data Class	5	Data Class	4	Long Method	4
	Data Clump	1	Shotgun Surgery	5	Shotgun Surgery	5	Divergent Change	4	Shotgun Surgery	4
	External Duplication	1	Controller Class	3	Divergent Change	3	Shotgun Surgery	4	Controller Class	3
	Internal Duplication	1	Divergent Change	3	ISP Violation	3	Class Global Variable	3	Data Clump	3
	Long Parameter List	1	ISP Violation	3	Duplicated Code	2	Method No Parameter	3	Divergent Change	3
	Message Chains	1	Complex Class	2	Long Parameter List	2	No Inheritance	3	ISP Violation	2
	Primitive Obsession	1	Duplicated Code	2	Temporary variable	2	No Polymorphism	3	Inappropriate Intimacy	1
	Schizophrenic Class	1	Long Parameter List	2	Data Clump	1	Spaghetti Code	3	Long Parameter List	1
	...		...		...		...			

few cloned lines does not necessarily, remove the LONG METHOD) and none of these papers detail the situations where the relation between LONG METHOD, LARGE CLASS and DUPLICATE CODE is relevant. Thus, we suggest that it is still necessary to identify situations where the relationship between DUPLICATE CODE and LONG METHOD is relevant because not all LONG METHODS are caused by DUPLICATE CODE and vice-versa. Similarly, we also consider important to investigate the relation between DUPLICATE CODE and LARGE CLASS because although the former may cause the latter, the latter is not always a consequence of the former.

LARGE CLASS. The literature relates LARGE CLASS to LONG METHOD and LONG PARAMETER LIST by volume/size metrics as follows: a) *"*LONG METHOD *is a method with a high number of lines of code and a lot of variables and parameters are used"* [311]; b)*"Consider a parameter list long when the number of parameters exceeds 5"* [355]. LARGE CLASSES are highly coupled to DATA CLASSES [356], which mostly present incoming dependencies from FEATURE ENVY methods [327]. Some studies relate LARGE CLASS to FEATURE ENVY [327, 375], and LONG METHODS to LARGE CLASS [323, 327].

Palomba et al. [354] proposes a new bad smell detection technique based on change history mining. This technique is especially suited to finding smells manifested in code changes. In fact, they propose a detection strategy based on past changes, i.e., they speculate that changing LARGE CLASS with LONG METHODS, which are often affected by FEATURE ENVY, suggests triggering changes in several areas of the system, which reveals the occurrence of SHOTGUN SURGERY. Another finding related to this relationship is that: *"the* GOD CLASS *and classes with* SHOTGUN SURGERY *were changed more frequently than the other classes"* [323].

FEATURE ENVY. As previously mentioned, this smell is mostly studied with other smells, especially with LARGE CLASS and/or LONG METHOD. This is expected, because these smells are related to *"how many responsibilities are implemented in an entity (method/class)"*. Thus, if an entity implements many responsibilities, the probability that this entity is mostly interested in other entities also grows, and this behavior is related to the definition of FEATURE ENVY. This type of relation was reported in [329, 375]. The smell SHOTGUN SURGERY can occur when a single responsibility has been split up among a large number of classes (symptom of code scattering [39]). Thus, responsibility scattering could potentially introduce FEATURE ENVY. However, this relation is unclear in the literature.

LONG METHOD. Liu et al. [296] report a scenario of possible relation between FEATURE ENVY and LONG METHOD. Its premise is that LONG METHODS result by the combination of bad smells (e.g., FEATURE ENVY and DUPLICATE CODE). In [327], they showed that the majority of the LARGE CLASSES also manifest LONG METHODS, and several of these LARGE CLASSES accessed data/methods from other areas of the system.

DATA CLASS. According to Yamashita and Moonen [327], most of the artifacts with DATA CLASS present dependencies with FEATURE ENVY methods. Pietrzak and Walter [375] report that in 92% of cases, the existence of DATA CLASS indicates the presence of FEATURE ENVY. Strategies to detect LARGE CLASS usually consider the size (e.g., LOC) and the occurrence of DATA CLASS [373]. According to Moha et al. [365], a LARGE CLASS is associated with several DATA CLASSES. Pietrzak and Walter [375] report that DATA CLASS suggests the existence of LARGE CLASS because DATA CLASS is related to FEATURE ENVY.

We observe that some meaningful relationships were not explicitly co-studied in the literature, e.g., Arcelli Fontana et al. [309] point out that *"*LONG METHOD *refers to methods that tend to centralize the functionality of a class"* and suggest that clients of a class that have LONG METHOD(S) probably do not need to invoke the additional methods of this class. In other words, clients are unlikely to invoke the inherited super-class methods [52]. Thus, we conjecture a possible relationship between LONG METHOD and REFUSED BEQUEST. Another interesting point to note is that smells are likely inter-related, especially, if they occur in LARGE CLASSES. In other words, if the size of code elements grows up, different kinds of anomalies related to those large structures will likely appear. In this sense, one would think of smells to better characterize anomalies in large structures, to use that characterization to assess the (negative) impact in products and process, and possibly guide the process for fixing those anomalies.

Karasneh et al. [370] investigated the relationship between quality of the UML design models and the source code. They report that, on average, the proportion of classes in design models and code with the same smells is 37%. This is an evidence that smells may appear early. This study was focused on seven types of smells but they do not consider the interactions between them. Thus, according to our classification, this study is a co-occurrence of smells. We would suggest that empirical studies considering the interactions between smells are needed, especially to investigate the relationship between design models and source code.

Lessons About RQ3.3

We observed that the most frequently studied bad smells (LARGE CLASS, FEATURE ENVY, LONG METHOD) co-occur in papers. However, only a small percentage of these co-occurrences are actual co-studies that investigate the interaction between them, suggesting that this is still an area deserving attention. The current studies on the co-existence of smells in code suggest an association with maintenance and design problems.

7.4 RQ3.4: Which are the most used tools for handling bad smells in the experimental setup?

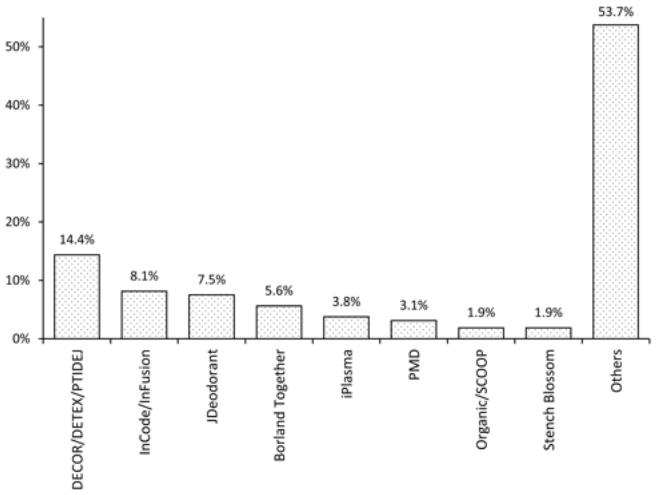

Fig. 6. Tools to handle bad smells (OBSG papers).

Fig. 6 presents the main tools/techniques used for handling bad smells, extracted from the papers. It is worth mentioning that some papers (e.g., [299, 312, 364]) use several tools/techniques because sometimes they perform experimental comparisons. In this case, each one is separately cataloged with its respective name (e.g., `JDeodorant`, `iPlasma`) and if this tool/technique appears in other papers, the number of occurrences is incremented. This procedure revealed that out of 82 tools/techniques, about 62 appeared in only one paper (e.g., `SMURF` [364], `Jmove` [392]) and 12 occur in two papers (e.g., `HistoryMiner` [351], `HIST` [354]). Thus, these were accounted in the *Others* group (see Fig. 6). *Others* represent 53.7% of the total occurrences. This finding demonstrates that the tools/techniques are greatly spread over the papers, reinforcing the suggestion that researchers are crediting bad smell inconclusive impacts to tools/techniques, yielding the search for new detection methods. Another argument is found in the discussion section of paper by Rasool and Arshad [417], who report that `PMD` and `JDeodorant` tools detect GOD CLASS partially differently if applied to the same source code. According to them, most of this disparity is due to the use of different metrics in the detection strategies.

In our study, we grouped the `DECOR`, `DETEX` and `Ptidej` occurrences in the same classification because, according to the `Ptidej`[6] team, "`DECOR` *is a method whose instances are detection techniques for code and design smells.* `DETEX`*, our instantiation of* `DECOR`*, allows the specification and the detection of defects such as code smells and antipatterns using a unified vocabulary and a dedicated language.* `Ptidej` *is the front-end to the tool suite for evaluating and improving the quality of object-oriented programs, reverse-engineering object-oriented programs (*`AOL`*,* `C/C++`*,* `Java`*), and promoting patterns.* `Ptidej` *integrates* `DECOR` *as well as visualization algorithms to ease the understanding of detected defects.*"

According to Fig. 6, `DECOR` (DEtection & CORrection [366]) is the most frequent tool, representing 14.4% of all occurrences. We suggest that the main reasons that yield this wide acceptance are:

1) the technique is widely documented in the scientific literature [53, 365, 366], allowing the comprehension of the internally used mechanisms;
2) it provides the creation of rules considering the relation/interaction among several bad smell types (see Fig. 4 in [366]); Yamashita and Moonen [372] report that one of the most desirable feature in tools handling bad smells is the customization ability (see Table V in [372]);
3) The tool is able to detect a higher number of distinct identifiable bad smells compared to other tools.

The second most frequent tools are `InCode/InFusion` (8.1%). These were proprietary tools developed by the same company (which does not exist anymore) and the difference is that `InFusion` was more comprehensive and offered more features. The two tools use the same bad smell detection mechanisms, therefore, we grouped their occurrences.

In next positions, we have `JDeodorant` (7.5%) and `Borland Together` (5.6%) tools. `Borland Together`[7] is a proprietary tool that allows to identify several bad smells (e.g., DATA CLASS, FEATURE ENVY, SHOTGUN SURGERY, ...) [323]. On the other hand, `JDeodorant`[8] is an open source tool that allows to identify five bad smells, namely: FEATURE ENVY, TYPE CHECKING, LONG METHOD, GOD CLASS and DUPLICATE CODE. Compared to `DECOR`, both tools are more restrictive concerning the composition of rules used to detect bad smells.

In this paper, we did not perform a systematic analysis of the tools/techniques listed before because there are already some studies about this [112, 417]. On the other hand, these studies do not rank the tools by the number of occurrences in papers as presented here.

To unveil the limitations of tools, we analyzed the available features in the most used tools (top-8). The features were extracted from the papers using those tools in empirical studies (e.g., [320, 350, 391]) and from papers that describe the tools (e.g., [31, 335, 342]); and whenever necessary, we also checked the implementation of the tool (e.g., PMD, Ptidej). Table 11 details the main features of the top-8 tools. For some features (represented by •), we were not able to check the presence of these features in some tools.

In the first group of features (*Detection Technique*), there are tools that aim to detect more than a fixed set of smells.

6. http://www.ptidej.net/research/designsmells
7. http://www.borland.com/en-GB/Products/Requirements-Management/Together
8. https://github.com/tsantalis/JDeodorant

TABLE 11
Features of top-8 tools (OBSG papers).

Type	Features Description	DECOR/Ptidej	InCode/inFusion	JDeodorant	Borland Together	iPlasma	PMD	Organic/SCOOP	Stench Blossom
Detection Technique	Machine learning based detection technique.				•				
	Configurable set of detected smells.	✓			•			✓	
	Fixed thresholds in detection rules.				•		✓	•	•
	Thresholds of metrics are available for customization.	✓	•		•			•	
	Thresholds of metrics are specified dynamically (e.g., boxplot).	✓	✓		•	✓		•	•
	Describe relationships between classes (e.g., aggregation).	✓		•	•				
Distribution & Scope	Form of distribution.	◎	◎	⊗	⊗	◎	⊕	⊗	⊗
	Supported languages:								
	Java,	✓	✓	✓	✓	✓	✓	✓	✓
	C++,	✓	✓			✓	✓		
	Other languages.		✓				✓		
Refactoring & Architecture	Support for Refactoring.			✓	•				
	Detection of architecturally relevant smells.				•			✓	
	Categorization of inter-related smells.				•			✓	
Usability	Interaction with smelly code.		✓	✓	•		✓	✓	✓
	Interactive visualization environment.		✓	✓	•				✓
	On-the-fly detection of smells while coding.				•				✓
	Metrics used are collected by external tools/JAR.			✓	•			✓	
	Analysis of multiple versions of a system.				•				
	Filtering capabilities (e.g., production or test code).				•				

✓ Feature avaliabled; ✓ Feature partially available.
• Information undetermined and/or unavailable.
◎ Standalone; ⊗ Plugin; ⊕ Standalone & Plugin.

So, they provide a DSL (*Domain Specific Language*) for the specification of detection strategies, enabling the selection of metrics and thresholds for each smell. On the other hand, we observed tools that are more rigid, where the thresholds and/or metrics used to detect the smells are not available for customization. In some tools, the thresholds of metrics are specified according to the system, e.g. using statistical techniques [31]. For the tools where the metrics are dynamically specified, it is also possible to customize thresholds, e.g. Moha et al. [335] describe parameter "fuzziness". The tool Ptidej can collect inter-method/class relationship data (e.g., aggregation). This feature is useful to specify rules used to detect smells (e.g., FUNCTIONAL DECOMPOSITION [335]). However, Ptidej does not collect intra-method relationship data and according to Hall et al. [357] this feature is helpful for detecting some smells (e.g., SWITCH STATEMENTS).

In the second group of features (*Distribution & Scope*), we noticed that the tools are distributed in plugins and/or standalone packages. There are tools designed to handle multiple languages, but the main language is Java.

In the third group of features (*Refactoring & Architecture*), we found only one tool that provide a mechanism designed to assist refactoring. In this case, the tool JDeodorant[9] supports Extract Method, Extract Class and Move Method refactorings. We also identified a tool that detects smells based on their influence on a specific aspect of the software. In particular, the tool Organic/SCOOP is designed to identify architecturally relevant smells using architecturally-sensitive metrics and strategies [54]. According to Oizumi et al. [320], this tool is also able to categorize the smells into four topologies.

The fourth group of features in Table 11 is related to usability. Most of the investigated tools allow developers to interact with smelly code, i.e., the tools provide the location of the detected smells, allowing developers to identify source code that require immediate refactoring, as well as dependencies between smells. This feature is partially present for the tool PMD, because the plugin version has this feature, but the standalone version does not. The next property is related to the visual information that supports smell understanding. In JDeodorant[10], each smell instance is visualized in the form of an enriched UML class diagram showing the dependencies between the class members involved in the code smell. This visualization allows developers to have a better understanding of the causes and severity of each code smell instance. Stench Blossom [342] provides an interactive visualization environment designed to give programmers a quick, high-level overview of the smells in their code, and then, if desired, to help understanding the sources of those smells. This tool also has on-the-fly detection of smells while coding. We also noticed that some tools do not collect all the metrics necessary to identify smells, for example: i) organic/SCOOP receives as input a metrics file in csv format [54]; ii) the tool JDeodorant relies on external clone detection tools for finding DUPLICATED CODE within a Java project [55]. The last two items of Table 11 are features unavailable in the top-8 tools that could improve their usability. For example, according to Macia et al. [54] the analysis of multiple versions of a system can identify critical smells and tend to improve the accuracy of conventional detection strategies.

We also classified the tools/techniques as follow: a) **Public**: the paper shares a link that allows to download the source code and/or a compiled version of the tool, and the hyperlink was still available to download in 2017; b) **Deprecated**: the same as Public, but either the hyperlink is broken or it does not exist; c) **Commercial**: tools that need payment to be used; d) **Ad-Hoc**: researchers do not share the implementations of these tools/techniques.

Below, we quantify these classes of tools:

- Public: we found 35 implementations (see Table 14):
  - 12 share the source code and some compiled version (e.g., `JDeodorant` [50], `DECOR/Ptidej` [366], `PMD`);
  - 11 share only the source code (e.g., `ChangeDistiller` [325], `HULK` [358], `Puppeteer` [415]);
  - and the remaining only share the compiled version (e.g., `SpIRIT` [338], `LBSDetectors` [399], `LAPD` [401]).
- Deprecated: we found 5 tools/techniques whose hyperlink does not work anymore (e.g., `InCode/InFusion`, `Organic/SCOOP`).
- Commercial: we found 4 tools (`Pascal Analyzer`, `Borland Together`, `Understand`, `SonarQube`).
- Ad-Hoc: we found 33 tools/techniques (e.g., `P-EA` [340], `HIST` [354], `OBEY` [388]).

Table 14 shows which tools/techniques are used by the papers to handle the bad smells. The association between tools/techniques and smells were extracted from the text in papers and is represented with "*X*" in each cell of Table 14. We use this strategy because there are tools/techniques that do not have a public documentation and/or do not have a public version. This means that some bias is possible: Fu and

9. https://github.com/tsantalis/JDeodorant

10. https://users.encs.concordia.ca/%7Enikolaos/projects.html

Shen [301] wrote *"For* SHOTGUN SURGERY*, we compare our approach with DECOR"*, however the current implementation of DECOR does not detect this smell. Another situation might occur in papers that use two or more tools to detect a code smell, these tools are able to detect different types of smells and the paper does not specify which smells each tool was able to detect [313]. We believe that the first situation is uncommon, probably because a previous version of that tool could actually detect the corresponding smell. The last condition is also unusual, most papers specify which smells each tool was able to detect [299, 323, 324, 357].

From Table 14, we can observe the distribution of tools between our classification (*Public*, *Deprecated*, *Commercial*, *Ad-Hoc*) and the bad smells. In general, our data show that 75.0% of smells can be detected by *Public* tools and 47.1% by *Ad-Hoc* tools. Considering only smells that occur in more than two papers (Smells listed in columns of Table 14 which are before the red line), we have 47 smells, and 72.3% of them can be detected by *Ad-Hoc* and *Public* tools. Moreover, with the exception of PARALLEL INHERITANCE HIERARCHIES, all other smells handled by *Ad-Hoc* tools are also handled by *Public* tools. On the other hand, there are 57 less studied smells (they appear in at most two papers). From them, 63.1% can be detected by *Public* tools and 24.5% by *Ad-Hoc* tools.

We can observe that empirical studies are typically conducted with the most common smells detected by the existing tools. However, there is a very high number of distinct smells which are already detectable by some tool, but some of these have not been extensively investigated in empirical studies, e.g., PROMISCUOUS PACKAGE occurs in one paper [359]. The literature also reports smells that have not been investigated empirically, e.g., Brown et al. [10] proposes the smell POLTERGEISTS — *"classes with very limited roles and effective life cycles. They often start processes for other objects"*. For this kind of smell, extensions on the available detection tools could be proposed to make empirical studies possible. For instance, the smell POLTERGEIST is found in classes with few responsibilities, with few commits during the life cycle, with most of the features being done by other objects, i.e., rules that seem reasonable to be automated.

We suggest that studies on these smells could better characterize their effective impact in software development and maintenance. Those studies could be even characterized as co-studies to investigate the interaction between that large set of distinct smells still poorly understood in the literature. Moreover, it is important to recognize that available tools still have some level of inaccuracy or some level of disagreement among them on how to detect smells. Thus, conclusions from empirical studies may also disagree depending on the tools used. A possible recommendation is that studies should consider detecting smells from more than one tool to compare the agreement level on the conclusions.

Lessons About RQ3.4

We observed a high diversity on the range of tools used in the experimental settings to handle smells. Many do not share their implementations, limiting their applicability. Moreover from the perspective of open science, these unavailable implementations hinder reproducibility and impose barriers to the underlying empirical studies, in particular for those aiming at comparing new approaches with the state-of-the-art. Our data also show that most of the smells can be detected by *public* tools. Finally, we observed that empirical studies are typically conducted with the most common smells detected by the existing tools.

7.5 RQ3.5: Which are the most frequent subject projects used in experimental evaluation?

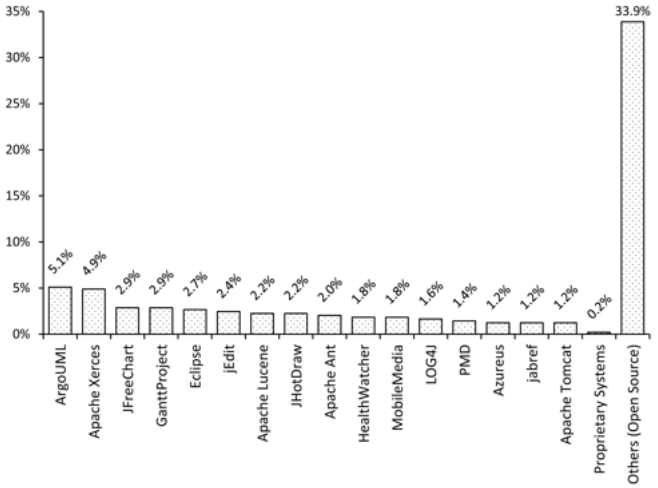

Fig. 7. Projects used in experiments (OBSG papers).

Fig. 7 presents the main subject systems used in the empirical studies presented in the analyzed papers.

The subject projects were also collected from manual analysis of the papers. These projects are cataloged with their respective names and the number of occurrences was incremented when a cataloged system appears in another paper. Similarly, when the paper uses closed source software, we increment the occurrences of *Proprietary Systems*. At the end, we found 226 distinct open source systems and found 18 occurrences of *Proprietary Systems*. From the total, 155 systems were used in just one paper, 39 systems were used in two papers, and another 15 systems occur in up to five papers. So, these 209 less frequently studied systems were grouped into the *Others* class, which represents 33.9% of the occurrences (see Fig. 7). We observe that there is a huge variability of subject systems being studied, and 63.5% of these are studied only once. This is an indicator that there is no widely accepted benchmark for bad smell studies and this can partially explain why there are some contradictory conclusions on the impact of bad smells.

In our study, which considers papers studying other types of bad smell, we found that 80.9% of papers perform empirical studies using open source systems, 14.8% use industrial/commercial systems with proprietary code (*Proprietary Systems*), and the remaining (4.3%) are subjective studies (e.g., interview, questionnaire, observation of developers). Most researchers perform empirical studies using open source software and we suggest that this

preference may be credited to the following factors: 1) simplicity to access information (source code, documentation, versions); 2) possibility to replicate the study conveying reliability/credibility; 3) community interest in the open source subject. In [6], the authors do not distinguish the type of analyzed source code.

The Java language is the most prevalent. The `ArgoUML`[11] system is present in 20.6% of analyzed papers and accounts for 5.1% of occurrences, being the most prevalent. The `Apache Xerces`[12] project is the second most recurring. It appears in 19.8% of analyzed papers and accounts for 4.9% of occurrences. The following systems are `JFreeChart`[13] and `GanttProject`[14]. Each one appears in 11.5% of analyzed papers and individually accounts for 2.9% of occurrences.

The seven top-ranked systems have at least three features that may partially explain the preference of the researchers: 1) they are large systems, widely disseminated in the open source community; 2) they are long lived projects with more than ten years, and stored in public repositories, allowing access to old versions of the code, which is relevant, as some bad smells (e.g., SHOTGUN SURGERY, FEATURE ENVY) are essentially studied over the history of the systems; 3) they are structured in subprojects, providing some evidence that the managers of these projects are concerned with using techniques for improving modularity and reuse, which is coherent with the concept of refactoring and minimization of the bad smell negative effects.

Lessons About RQ3.5

As previously mentioned, the reports on the impact of bad smells sometimes contradict each other. These contradictions may be credited to the wide range of tools and subject systems used in the experimental settings, suggesting that the lack of well-designed benchmarks should be addressed. The benchmarks could be constructed by searching for systems having the same characteristics as the most used systems.

8 RESULTS ON RESEARCHERS (TA4: who)

The next subsections are aimed at establishing the relationships among researchers with respect to research groups, and according to their interest in the different bad smells.

8.1 RQ4.1: How is the research community grouped around the types of smells? Do researchers study a broad and diverse set of bad smells, or concentrate on one or a few bad smells?

Fig. 8 shows the distribution of the distinct 530 authors in two groups: DCG and OBSG. There are 315 nodes (59.4%) in the DCG group, which corresponds to authors that published papers related **only** to DUPLICATE CODE (grey nodes in Fig. 8). The nodes in the group OBSG are classified into five categories:

11. http://argouml.tigris.org/
12. http://xerces.apache.org/
13. http://www.jfree.org/jfreechart/
14. https://sourceforge.net/projects/ganttproject/

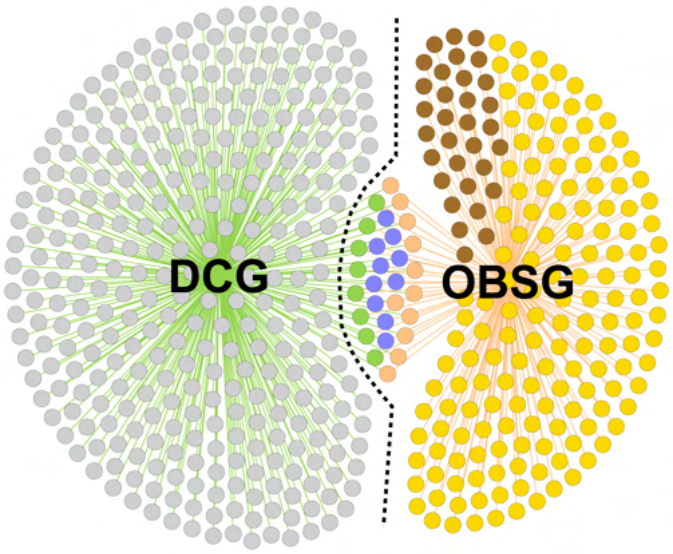

Fig. 8. Clusters of authors by paper groups (DCG/OBSG).

1) Yellow nodes: authors who DO NOT investigate DUPLICATE CODE;
2) Brown nodes: authors who study DUPLICATE CODE together with other smell(s) but did not publish papers in group DCG (e.g., co-occurrence of FEATURE ENVY and DUPLICATE CODE);
3) Green nodes: authors who have papers in both groups (DCG/OBSG), but mostly in DCG group;
4) Orange nodes: authors who have papers in both groups (DCG/OBSG), but mostly in OBSG group;
5) Blue nodes: authors who have papers in both groups (DCG/OBSG), in similar proportion.

There is only a small fraction of 29 authors (5.5%) in the intersection (Green, Blue and Orange).

It is interesting to note that DUPLICATE CODE has to be considered as a particular category of smell, not only because it is the oldest being studied, the one with large number of papers and the one being studied mainly alone, but also because the research community working on DUPLICATE CODE is largely separated from the community studying the other types of bad smells. Indeed, there are just a few authors working on these two universes. One possible reason is that the interest in DUPLICATE CODE emerged much before the other bad smells had been characterized, so its community has been nurtured for a longer time. Another possible explanation for this is that DUPLICATE CODE is quite versatile and does not heavily depend on internal or external factors of software. Also, the detection of this bad smell is not influenced directly by the existence of other types of bad smells (e.g., LARGE CLASS) and this could also have contributed to the observed behavior.

We observe that the OBSG community would benefit from more interaction with the DCG community. For instance, consider the study of genealogy of CLONES, which describes how groups of CLONES change over multiple versions of a system [219]. There are several DCG papers on this topic [160, 181, 207]. Even if there are some papers that studied the life cycle of smells (e.g., [308, 321, 360]),

to the best of our knowledge, there was not an intentional reuse of the methodological framework already developed for DUPLICATE CODE (CLONE).

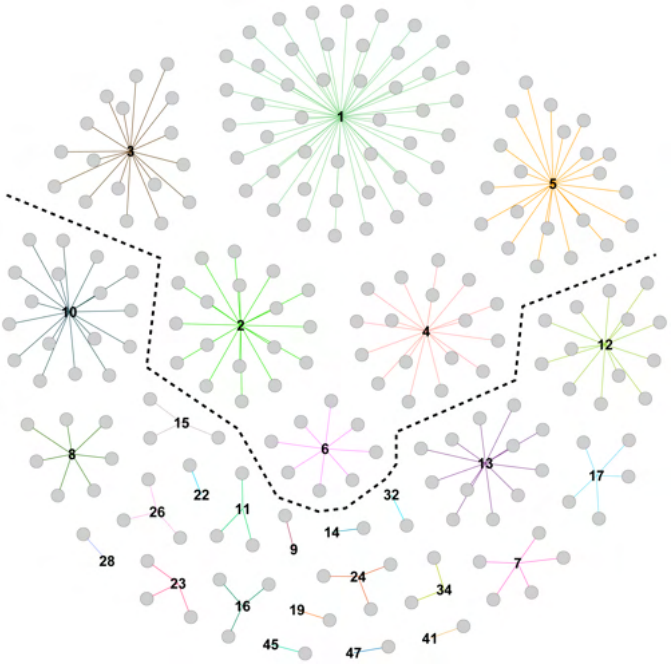

Fig. 9. Amount of bad smells studied by each author in OBSG papers.

To answer the second part of RQ (if researchers study a diverse set of smells or concentrate on few smells), we analyze the authors of OBSG papers. In this group, there are 215 distinct authors, who were grouped in Fig. 9 according to the number of distinct smells. From this figure, only 88 authors (40.9%) studied more than six smells and these are distributed into 22 clusters at the bottom of Fig. 9. 46 authors, a representative fraction (21.4%), studied only one bad smell.

Although the authors of OBSG papers have a reasonable interest in studying several bad smells types, the bad smell diversity is somewhat limited. This may be explained by the fact that the bad smells studied the most represent only a small set of the whole universe of proposed smells (e.g., LARGE CLASS, LONG METHOD, FEATURE ENVY), and still, they are likely to be studied in the same papers, as shown in Table 4.

Lessons About RQ4.1

Authors have different levels of interest in bad smells and few of them try to study smells that are apparently unrelated or tackle studies with a wide range of bad smells. We observe again that DUPLICATE CODE represents a smell on its own which deserves a separate and different type of literature review.

8.2 RQ4.2: Who are the researchers mostly interested (by number of papers) to the area of bad smells? Which were the countries and universities where bad smells studies have been conducted?

This research question focuses on elucidating which researchers have contributed more to specific bad smell topics.

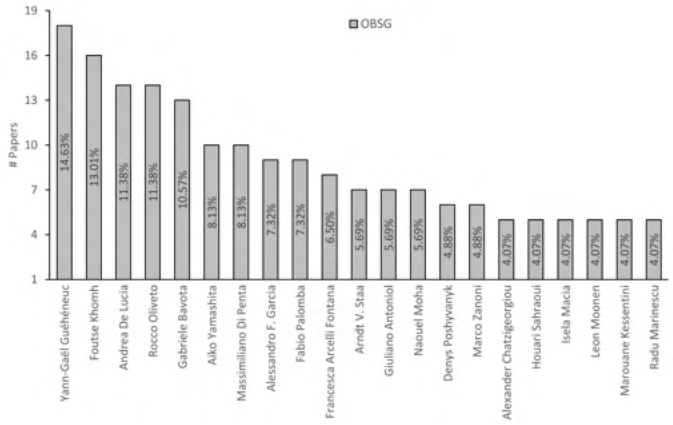

Fig. 10. Number of papers by author (OBSG papers).

The goal is to facilitate information access and to promote collaboration among researchers with similar interests.

A required disclaimer is that this question does **not** aim at ranking the most productive researchers. Researchers may work on different research lines other than bad smells. In our survey, we did not analyze every publication from researchers; therefore, it does not make sense to compare the productivity of researchers with these data.

Analyzing the OBSG papers, the main authors are: Yann-Gaël Guéhéneuc[15] (14.6%), Foutse Khomh[16] (13.0%), Andrea De Lucia[17] (11.3%), Rocco Oliveto[18] (11.3%), Gabriele Bavota[19] (10.5%). Fig. 10 summarizes this information. In general, the top-5 currently play the role of advisor or principal investigator.

The second part of this research question is aimed at pointing out institutions with more interest in bad smells, thus elucidating where specific kinds of investigation have been carried out. Even if the number of published papers is a reasonable proxy measure for level of interest in a theme, there could be other factors that would characterize the interest that were not considered.

The researchers more concerned in other bad smells and/or in the combination of them (OBSG papers) are affiliated to the following institutions: Polytechnique of Montréal[20], University of Salerno[21], University of Montréal[22], University of Sannio[23], and University of Molise[24] (see Fig. 11).

Data also reveals that Montréal (Canada) is the main site concentrating researchers interested in studying other bad smells, as well combinations of them. Specifically the institutions: Polytechnique of Montréal and University of Montréal are the main representatives in this area (Group OBSG).

15. http://www.yann-gael.gueheneuc.net/Work/Info/
16. http://www.khomh.net/
17. http://docenti.unisa.it/andrea.delucia
18. http://docenti.unimol.it/index.php?u=rocco.oliveto
19. http://www.inf.usi.ch/faculty/bavota/
20. http://www.polymtl.ca/
21. http://web.unisa.it/en
22. http://www.umontreal.ca/english/
23. http://www.unisannio.it/
24. http://www.unimol.it/english/

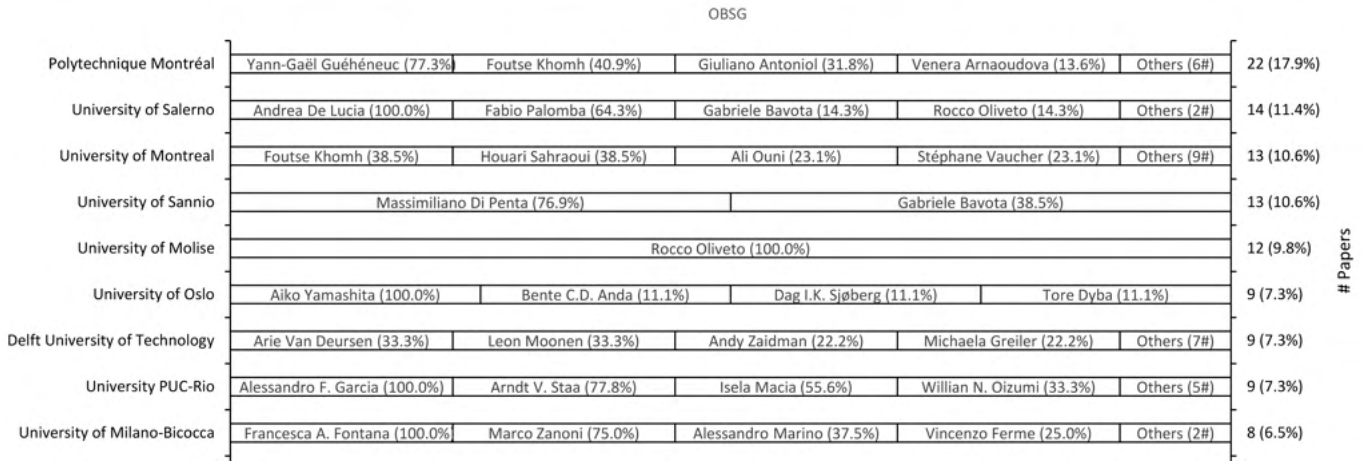

Fig. 11. List of affiliations on OBSG papers.

Our dataset shows that Canada (18.9%), USA (18.4%), Brazil (9.0%) and Italy (8.0%) are the most representative countries for the number of authors (see details in Fig. 12). With respect to the number of research centers, this sequence is a little different: USA (22.2%), Germany (11.1%), Canada (9.8%), Italy (7.4%) and Brazil (6.1%), see details in Fig. 13. The same analysis, but in terms of the number of publications, reveals another ranking: Canada (35), Italy (29), USA (24), Netherlands (12), Norway (12) and Brazil (11). The main research centers and their main authors are detailed in Fig. 11, where we use the number of papers to sort the research centers.

Analyzing Fig. 11, we observed that some authors move between the institutions or are affiliated with various institutions. In some cases, the collaboration with an institution occurs before the author becomes affiliated to that institution.

Another remark is that the universities are largely responsible for conducting research on the bad smells. Considering all authors of OBSG papers, only eight (4.4%) are not affiliated to universities.

Lessons About RQ4.2

We reported the main authors interested in the bad smells and their countries/affiliations. This information helps to monitor and track the progress and/or scientific trends and it is especially useful for newcomers to this field. We also observed that universities are largely responsible for conducting research on bad smells.

8.3 RQ4.3: How are the authors and their research groups interconnected? Does this interconnection impact on publications?

Social network analysis (SNA) is a suitable framework to answer this research question. According to Farine et al. [56], *"social network analysis is a tool for studying the social organization of groups based on the associations or interactions between individuals"*.

In SNA, individuals are represented by nodes and the relationships between them are represented by lines connecting the nodes. These diagrams are very useful to reveal group structures that are hidden (e.g., stars, alliances, subgroups) [57]. In this work, we used `Gephi` [58], an open source software that allows the creation, handling, and representation of these SNA graphs.

In order to answer this research question, we rely on Fig. 12, where the authors are represented by nodes. The line connecting two nodes means that both authors co-authored one or more papers. The size of each node represents the number of papers published by the author in the corresponding group (OBSG papers). Analogously, the line thickness connecting the nodes represents the extent to which the authors are publishing together. Finally, the node colors represent the country of the research institutions which the author is affiliated with. If an author published papers with different affiliations and the countries of the institutions are different, affiliation and country of the most recent paper is considered in the analysis. We also investigated the movement of authors and we observed that 19 (8.8%) out of 215 distinct authors in the OBSG papers moved between universities (institutions). However, if we consider moving between countries, only 9 (4.2%) moved from one country to another.

Fig. 12 shows every author of OBSG papers and we observe the existence of several groups of different sizes. A group is characterized by a cluster, which can be isolated or have an interface with other groups (see Fig. 12). Most of the groups are comprised of authors that do not interact with other groups, i.e., clusters are more well-defined. Moreover, when interaction occurs, typically, the main/large node in a group (cluster) is the one that interacts with other groups. Actually, this also may explain why those nodes are larger. Most clusters are one-colored, indicating that groups are likely to be restricted to the same country, i.e., geographic localization is an important factor to determine collaboration.

We also observed that the groups are quite heterogeneous, in terms of number of papers. Some groups are exclusively composed of authors having just one paper or a

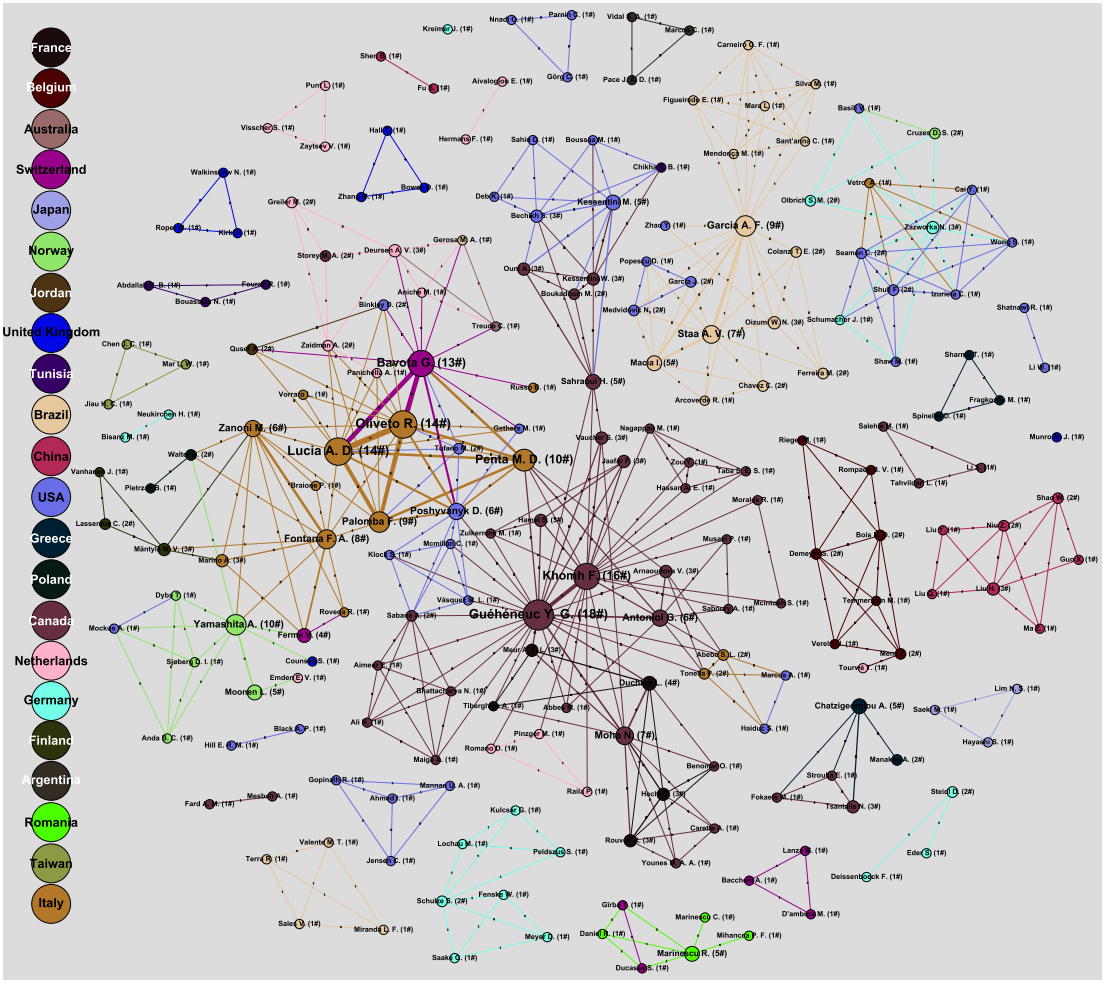

Fig. 12. Social Network Graph of authors in OBSG papers.

small number of papers. On the other hand, other groups are composed of authors having a moderate or higher number of papers. There are also more heterogeneous groups, i.e., groups having authors with a high variance in the number of papers, suggesting different roles of the researchers, e.g., advisor/PI or student. In general, Fig. 12 gives an idea of how future collaborations will look like, e.g., newcomers (students) will emerge from the interaction with authors (advisors) having a moderate or high number of papers.

The second part of this research question aims at clarifying if the interconnections among the authors have an effect on publishing. We calculated the correlation coefficient between the number of papers of authors and their corresponding number of different co-authors. We use Pearson correlation due to the relatively large size of data (df=212) [59]. For authors in the OBSG group, we have a relatively high Pearson's correlation: 0.848805, p-value<$2.2e{-}16$, df=212, IC 95% (0.805813 0.882896). Although this result was expected, we show it to reinforce the role of collaboration.

We also investigate if the co-authorship graph (Fig. 12) has the properties of a "small world graph", in which most nodes are not neighbors of one another, but the neighbors of any given node are likely to be neighbors of each other and most nodes can be reached from every other node by a small number of edges. According to Telesford et al. [60], for disconnected graphs, small world analysis can be done on the largest connected component of the network. Fig. 12 contains a large connected component along with many smaller disconnected components that vary in size. This largest component contains around 41.9% (90 of 215) of all authors and they have 246 edges between them. The study by Telesford et al. [60] reports on a quantitative approach for identifying a small world graph. In their approach, the clustering coefficient (C) and short path lengths (L) of the large connected component are measured against those of their equivalent derived random networks (C_{rand} and L_{rand}). Next, these measures are used to calculate the small-coefficient (σ). They report that a small world graph is a graph with: $C \gg C_{rand}$, $L \approx L_{rand}$ and $\sigma > 1$. For the large connected component of Fig. 12, these conditions are met ($C = 0.805$; $C_{rand} = 0.071$; $L = 3.743$; $L_{rand} = 3.075$; $\sigma = 9.30$). We conclude that this large connected component is a small world graph, where the collaboration is flourished, and they are responsible for most of the OBSG papers (69 out of 124), having a higher number of papers per author. The disconnected 30 sub-graphs represent smaller groups with a lower number of papers, in general, but collectively they produce a significant number of OBSG papers (55 out of 124). This division of the papers between

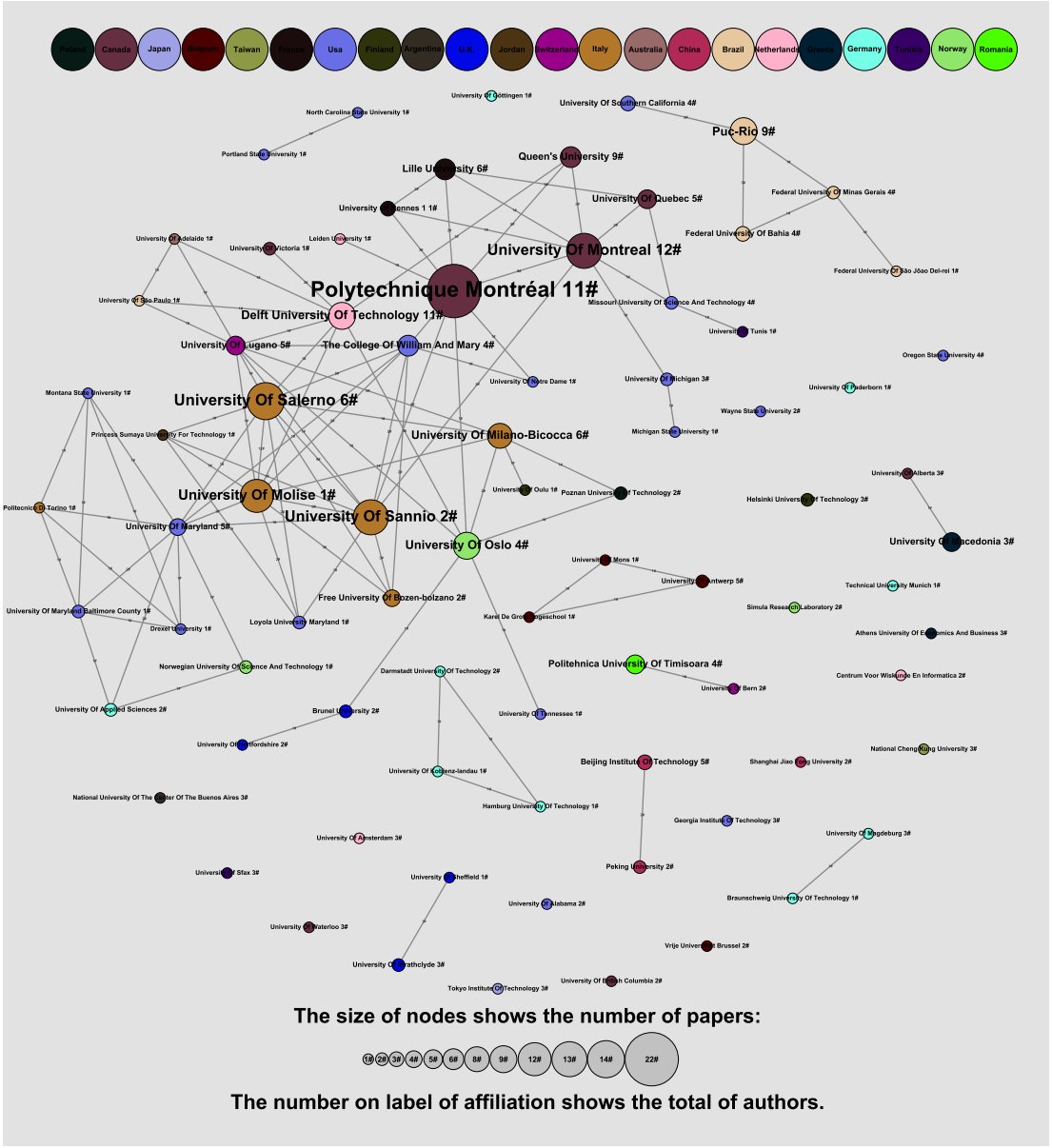

Fig. 13. Social Network Graph of affiliations in OBSG papers.

a large connected group and many isolated groups unveils the dynamics of how the findings are being produced.

In order to complement the analysis of Fig. 12, we also created Fig. 13 to represent the interconnections between research groups. This figure is like the previous one (Fig. 12), however, the universities (institutions) are represented by nodes and the colors show their countries. The node size still represents the number of papers, and the score inside the labels shows the number of authors with that affiliation — in this case, we consider the affiliation of authors at the time of publishing. In general, this new figure gives an idea of how committed are the research centers on the bad smells and how much they are working together.

Analyzing Fig. 13, we observe that some research centers are highly interconnected to others, e.g., University of Maryland (10), University of Molise (10), University of Salerno (10), University of Sannio (10), Polytechnique of Montréal (9), University of Lugano (9). However, the density of interconnections does not always correspond to the number of papers (size of nodes), e.g., the authors affiliated with the University of Maryland published four papers. On the other hand, many of these interconnections reflects the interest on bad smells, e.g., the University of Salerno and the Polytechnique of Montréal shows fourteen and twenty-two papers, respectively. So, the institutions are reasonably different and, possibly, this is reflected in the number of authors (score inside the label of nodes), e.g., the University of Salerno showed six authors and the Polytechnique of Montréal had eleven authors.

Lessons About RQ4.3

The scientific connections among researchers (observed by SNAs) seem to exert some influence on publishing, including defining those bad smells that are subject

of studies. The community is organized into a large *small world graph* of collaborations that is responsible for most studies. Moreover, there is a large number of disconnected groups that are responsible for a substantial part (44%) of the studies. We observe a concentration on some geographic locations and/or affiliations, which can be explained by the small world graph of collaborations.

9 Results on the Distribution of Papers Among Venues (TA5: where)

This section quantifies the venues that are more prone to publish bad smell studies.

9.1 RQ5.1: Are there venues more inclined to publish papers on a particular set of bad smells?

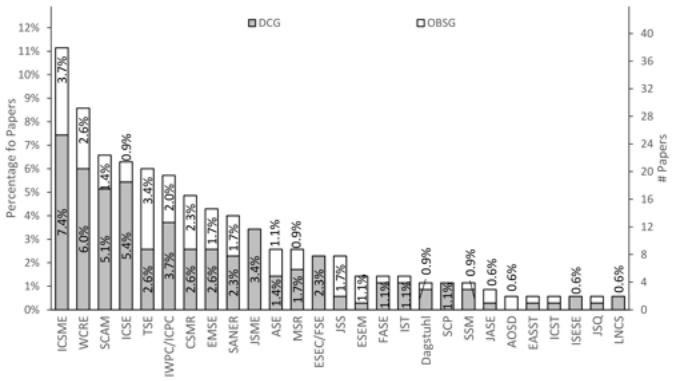

Fig. 14. Venues distribution by groups (OBSG/DCG).

To answer this question, the proportion of indexed papers in the main venues over more than one decade (2002 — 2017) is presented in Table 12. This table considers only OBSG papers and presents venues with three or more publications. The 13 venues presented in this table are responsible for 69.3% of the OBSG papers. In order to complement the analysis and to provide an overview of our dataset, Fig. 14 considers all papers of the *Final Database* and the dataset is organized into two categories (OBSG/DCG). The 27 venues presented in this figure are responsible for 82.3% of the papers comprising the *Final Database*. This figure shows that there are some venues where Duplicate Code papers are almost as frequent as all other bad smells together. For instance, considering the top-15 venues, ICSME, WCRE, SCAM, and ICSE have a higher proportion of papers on Duplicate Code compared to all other bad smells together. Other venues in the top-15, where the proportion of Duplicate Code is high (more than 1/3) compared to all other bad smells are ICPC, CSMR, EMSE and MSR.

Numerically, the results show that ICSME is the main scientific vehicle to disseminate knowledge about bad smells (see Table 12 and Fig. 14). In 2004, the first papers about other bad smells appeared [339, 371]. According to Table 12, the years 2012, 2013, and 2016 were for ICSME the most representative in the series with 53.8% (7) of the papers on bad smells.

In the second position, considering only OBSG papers, we have the TSE journal with 12 papers (see Table 12). In comparison with ICSME, first papers published in TSE appeared later (2007), as expected; in terms of number of papers, the years 2013 and 2014 were the most representative for TSE.

Following, we have WCRE with 7% of the papers comprising the group OBSG (see Table 12). The earliest study considering other bad smells [379] appeared in 2002. Similarly to ICSME, WCRE was dominated (70.0%) by Duplicate Code publications (see Fig. 14). Generally, the number of papers published in this venue increased (see Table 12).

In 2014, WCRE joined CSMR and in this year the papers were published in the proceedings of CSMR-WCRE, while in 2015 CSMR-WCRE was named International Conference on Software Analysis, Evolution, and Reengineering (SANER). Thus, to avoid data superposition, since 2014 WCRE and CSMR papers were separately classified (SANER). It is interesting to note that if considered together, CSMR, WCRE, and SANER would be the venue with the largest number of papers.

In order to explain why researchers publish more in ICSME and WCRE[25] than in other conferences (e.g., ICSE, SCAM), we analyzed the *"Program Committee"* composition of the most representative editions (2010 — 2016). These years were more representative because they represent 79.0%, 98 out of 124 papers comprising the OBSG group (see Fig. 2). Combining this information with the researchers and their inter-connections, we observed that the committees of the main venues (ICSME/WCRE) include many more researchers with an interest in bad smells than other conferences and generally, most of these researchers are points of interface within their research groups (see details in Subsection 8.3). So, it is natural that these venues gain visibility from the interested community and researchers are stimulated to publish there.

We observed that most venues are interested in bad smells regardless of its type. Exploring the *Research Tracks* of the main venues, we observed that the call for papers usually has generic *topics* (e.g., *Software refactoring, restructuring, renovation, migration and reengineering* at ICSME; *Program transformation and refactoring* at WCRE) and, in some cases, there are bad smell specific *topics* (e.g., *Code cloning and code provenance* at ICSME) coexisting with generic *topics*. With respect to specific *topic* of bad smells, the most common is related to *clones*. This also demonstrates that Duplicate Code smell is a more mature and specific topic, unlike other smells. Also, the co-occurrence of specific and generic *topics* on bad smells would explain why the community working only on Duplicate Code and the community working on the other types of smells are largely separated.

According to the data, 2013 was the year with the largest number of venues (15) simultaneously publishing papers in the bad smell area, and we can observe that this is the year with the largest number of papers in the main venues. This would suggest a growing interest in the topic. This fact also explains the peak of 42 papers observed in Fig. 2.

25. We also consider CSMR-WCRE and SANER.

TABLE 12
Venues distribution (OBSG papers).

Venues \ Years	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	Total
ICSME			2 (15.4%)				1 (7.7%)		1 (7.7%)		2 (15.4%)	2 (15.4%)	1 (7.7%)	1 (7.7%)	3 (23.1%)		13 (10%)
TSE						1 (8.3%)		1 (8.3%)	1 (8.3%)		1 (8.3%)	3 (25.0%)	3 (25.0%)	1 (8.3%)	1 (8.3%)		12 (10%)
WCRE	1 (11.1%)							2 (22.2%)			3 (33.3%)	3 (33.3%)					9 (7%)
CSMR		1 (12.5%)	1 (12.5%)	1 (12.5%)						2 (25.0%)	1 (12.5%)	2 (25.0%)					8 (6%)
IWPC/ICPC					1 (14.3%)					1 (14.3%)			2 (28.6%)		3 (42.9%)		7 (6%)
EMSE					1 (16.7%)						1 (16.7%)			1 (16.7%)	3 (50.0%)		6 (5%)
JSS						1 (16.7%)				2 (33.3%)	1 (16.7%)	1 (16.7%)		1 (16.7%)			6 (5%)
SANER														1 (16.7%)	3 (50.0%)	2 (33.3%)	6 (5%)
SCAM										2 (40.0%)		1 (20.0%)		2 (40.0%)			5 (4%)
ASE												1 (25.0%)		1 (25.0%)	2 (50.0%)		4 (3%)
ESEM									1 (25.0%)	1 (25.0%)				2 (50.0%)			4 (3%)
ICSE												1 (33.3%)		1 (33.3%)	1 (33.3%)		3 (2%)
MSR												1 (33.3%)		1 (33.3%)	1 (33.3%)		3 (2%)

Lessons About RQ5.1

Some venues, like ICSE, SCAM, ICSME, WCRE, ICPC, CSMR, and EMSE, have a higher proportion of DCG papers, being highest in ICSE and SCAM. Since 2004, the interest in other bad smells has increased, where TSE and ICSME are the venues with highest proportion of studies.

We also observed that the call for papers of venues might in part explain why community working on DUPLICATE CODE is largely separated from the community studying the other types of bad smells, having indeed a specific topic of interest.

TABLE 13
Most discussed metrics on OBSG papers.

Frequency	Dimension	Example of Metrics
High	Size	Lines of Code (LOC), Number of Classes (NOCS).
High	Complexity	Cyclomatic Complexity (CYCLO), Weighted Methods Count (WMC).
High	Cohesion	Tight Class Cohesion (TCC), Lack of cohesion in methods (LCOM).
High	Coupling	Coupling Between Objects (CBO), Access to Foreign Data (ATFD).
Medium	Lexical	Vocabulary of Method Name (VMN), Vocabulary of Field Name (VFN).
Medium	Inheritance	Number of children (NOC), Depth in inheritance tree (DIT).
Medium	Encapsulation	Number of Public Attribute (NOPA), Number of Accessor Methods (NOAM).
Medium	Structural	Static Method (SM), Abstract Class (AC), Overridden Method (OM).
Low	Polymorphism	Number of Potentially Polymorphic Instructions (NPPI).
Low	Performance	Number of Garbage Collection Calls (NGCC), Memory Usage (UM), Number of Delayed Frames (NDF).
Low	Design Pattern (MVC)	Number of Routes (NOR), Number of Services as Dependencies (NSD), Non-Framework Response for a Class (NFRFC).
Low	Change Information	Number of Methods Committed (NMC), Number of Classes Modified (NCM).
Low	Architecture-Sensitive	Number of External Elements (NEE), External Fan-out (EFO).
Low	Developer IDE Activity	Number of Viewed files (NVF), Time Used For Editing (TUE).
Low	Textual	Commit Messages Analysis, Comment Messages Analysis.
Low	Design Models (UML)	Number of Smelly Classes in Models (NSCM).

Frequency on OBSG papers: High ≥ 5; $3 \leq$ Medium ≤ 4; $0 \leq$ Low ≤ 2.

10 DISCUSSION & FUTURE DIRECTIONS

We have shown that although a large body of knowledge has been produced regarding the detection of smells and respective association with maintenance factors, there is still some surprising and contradictory results. In this section, we provide a perspective on how smell detection and monitoring can be further investigated to drive towards more convergent and actionable evidence.

10.1 On the improvement of smell detection with new kind of metrics

Metrics are largely used to define rules for smell detection in code entities (e.g., class/method). Several metrics have been implemented, e.g., the PADL framework implements more than 60 metrics [334], and many of them are used to detect smelly codes [352]. Table 13 shows the most recurrent metrics on OBSG papers organized according to sixteen dimensions based on previous work [309]. The most frequent dimensions are related to the most studied smells in OBSG papers (Table 4). Typical approaches compute these metrics from the source code of one snapshot of the system (e.g., [305, 306]). However, over recent years, the metrics also have been collected from version-control systems (e.g., SVN, Git) [301, 354, 361], which still allows extracting metrics for a specific snapshot (revision), but also allows extracting other kind of metrics (e.g., classes added/removed/moved/renamed), i.e., the *Change Information* dimension. Metrics can also be extracted from UML diagrams [370]. However, generally these diagrams are rarely included in version control systems together with source code, so extracting metrics from UML diagrams would first require reverse engineering them from source code.

Future Direction: We suggest that metrics from other sources could be investigated to improve the quality of current detection strategies. As examples of new sources, we suggest to explore the less studied metrics that can be obtained from different kind of repositories: (i) developer profile (e.g., expertise on specific components); (ii) development environment (e.g., error/faults on integration scripts); (iii) UML diagrams (e.g., class diagrams); (iv) communication process (e.g., time used to report/discuss issue(s)).

10.2 On the use of evolution metrics for smell monitoring

We observed that most of the detection approaches (see Fig. 5), define composite rules based on metrics (e.g., GOD CLASS: a class has high AOFD (Access Of Foreign Data) and WMPC (Weighted Method Count) and TCC (Tight Class Cohesion) [306]). To define what is *"high"*, two strategies to define thresholds for the corresponding metrics have been proposed: thresholds can be *constant* (e.g., [357, 405]) or *variable* according to a predefined criterion, for example, project-specific thresholds (e.g., [339, 364]). Over recent years, we observe that *variable* thresholds have received greater atten-

tion, and many techniques to define those thresholds have been proposed, e.g., Maiga et al. [364] use the statistical theory of supervised learning (SVM); Moha et al. [366] use the classical statistical methods (inter-quartile ranges) and Khomh et al. [373] use conditional probability (Bayesian Belief Network). The hypothesis that constant thresholds may be too restrictive to cope with a wide range of specific contexts may be an indication that instead of detecting smells on the current revision to decide if refactoring is required or not, a more continuous approach of monitoring the evolution of smell-related metrics may play an important role in taking more informed decision on the necessity of refactoring.

Bavota et al. [308] detect the smell COMPLEX CLASS with the rule: *"a class having at least one method for which McCabe cyclomatic complexity is higher than 10"*. Their results showed the lack of interest of developers in refactoring COMPLEX CLASSES, because that would be very challenging. Based on this observation, maybe developers or software engineers are interested in monitoring the evolution to prevent the introduction of these kinds of bad smells, rather than fixing them.

A recent study [61] has shown that, in general, smells are long-lived and those that are removed, tend to be removed just after the insertion (after ∼10 commits). So, there is some indication that developers might be especially interested in monitoring the introduction of anomalies.

Future Direction: We suggest that rules for smell detection could also rely on the evolutionary tendencies of metrics, e.g., *the evolution of the McCabe metric on the different versions of the system could be monitored to inform a timely configuration for refactoring*. Although, some strategies to detect smells from the version history have been proposed (e.g., `HIST` [354]), to the best of our knowledge no study has investigated/considered the evolution of metrics (e.g., increase/decrease) to construct rules based on variable threshold to detect bad smells (e.g., LONG METHOD: *all methods having LOC higher than the average of the system and LOC being increased over the releases*).

10.3 On the human perception of bad smell

Some papers map or evaluate the subjective human perception of bad smells, e.g., [162, 312, 313, 326] (see Subsection 7.1). These papers concentrate on two groups of interest: *Developers* and *Students*. One interesting fact is that although a bad smell is detected in the source code, the developer may not agree that it is indeed a smell, and even more, he/she may not agree on the convenience of restructuring the code, posing a major challenge for detection and recommending tools.

An interesting issue may raise on the nature of bad smells. From a practical point of view, interesting bad smells are those that can be automatically detected. The detection process is a binary classifier: code components are classified as smelly or not. This dichotomy is used strictly in most studies on bad smells. Once an element is considered smelly, it typically carries all properties inherent to the corresponding bad smell(s). There is a recent study [352] that re-characterizes the dichotomous nature of bad smells to consider the intensity of bad smells, in order to improve bug prediction.

This raises the question that the association of an element with a design flaw that is convenient or even feasible to be restructured is much more complex than the dichotomic answer of bad smell detection. Nonetheless, the quantitative models already proposed for detection are an important asset to find and prioritize the relevant design flaws that are likely to be restructured.

Future Direction: We suggest that detection strategies evolve from a binary classification technique to a new environment for informed decisions about sub-optimal code that must be restructured, reporting the rationale, risks, consequences and benefits of that restructuring. In other words, this view is coherent with the previous directions, where we need a more holistic interpretation of sub-optimal code that is grounded on several aspects of development, including a retrospective and prospective analysis of metrics of interest.

10.4 On the necessity of representative benchmarks

Pate et al. [6] also show that `ArgoUML` is the most analyzed system, occurring in 26.6% of papers (8 out of 30 studies). Pate et al. [6] also reports a large diversity of open source systems used in empirical studies (in 30 papers, they found 51 different systems). We found 226 distinct open source systems and 18 *proprietary systems*. Note that Pate et al. [6] do not distinguish between open source and *proprietary systems*, and their work considers only CLONES. Nevertheless, both our and their data confirm the lack of a representative database containing source code and documentation, which could describe where and how bad smells could be fixed. This database would contribute toward conducting reproducible studies. As observed by Pate et al. [6], *"this makes clear the need for comparative studies in which the following variables are controlled: subject system, tracking interval, and tracking duration"*.

Moreover, we observed that apparent contradictions on research findings that we have reported previously are mostly consequence of the known threats to external validity. This is somehow expected because whenever some research finding is valid for some subject systems, different conclusions can be expected in studies using different systems.

Future Direction: The creation of a representative benchmark is very laborious to construct and would require a coordinated joint effort in order to be successful. In this direction, Palomba et al. [62] propose an open dataset (*Landfill*[26]) with 243 instances of five types of smells identified from 20 open source projects. Recently, Tufano et al. [351][359] improved this dataset to consider other smells (e.g., EAGER TEST) and they also included other open source projects (e.g., HSQLDB). However, the current dataset is restricted to a few kinds of smells (six code smells and two test smells). We also observed that Landfill does not permit discussion on how to refactor the instances of smells.

As an additional effort in the direction of robust benchmark, we suggest enhancing the promotion for adoption of open science practices, such as, the publication of datasets and artifacts used in empirical studies would provide in the short-term a large amount of data that would permit

26. http://www.sesa.unisa.it/landfill/

replications to better understand the influence of the context in the reported findings. Moreover, data availability would eventually permit the integration of different datasets into a large scale curated benchmark.

10.5 On the concept of smell lineage

Pate et al. [6] report four approaches for constructing clone lineage (or similarly, fragment traces) in terms of evolution. In one of these approaches, the clones are detected in all source code versions of interest, and then corresponding clones in consecutive versions are retroactively linked. Analyzing papers that use this type of technique (e.g., [6, 181, 219]), they map patterns that can model the appearance, disappearance, or reappearance of a clone between versions of code (e.g., a removal/change followed by an addition/change operation could cause another code fragment to reappear as a clone).

Considering the other types of smells (e.g., FUNCTIONAL DECOMPOSITION, SPAGHETTI CODE), in recent years, we have observed that some papers (e.g., [361, 360]) use the concept of lineage. Tufano et al. [360] investigated when developers introduce bad smells. They use multiple repositories and analyze each repository (r_i) using a *HistoryMiner*. This tool mines the entire change history of r_i, checks out each commit in chronological order, and then runs their *Ad-Hoc* implementation of a smell detector (*HistoryMiner* — similar to DECOR). As output, the tool produces for each source code file $f_j \in r_i$ the list of commits in which f_j has been involved, specifying if f_j has been added, deleted, or modified and if f_j was affected, in that specific commit, by one of the five considered smells. So, they can say when bad smells are introduced. However, they do not consider that the snippets of code can be moved from one entity to another, for example: the method A was in class C_1 on release R_1 and on the next release R_2 this method was moved to class C_2. This suggests that the smells could also move between entities. Therefore, distinguishing the cases where a smell was removed and a new one was added from cases when an entity was moved/renamed/created is important to construct a good lineage of smells. On this direction, Palomba et al. [354, 361] propose the tool *HIST* that considers this limitation. However, this tool is *Ad-Hoc* (the researchers do not share the implementation) and this limits its use by the community. Additionally, in these papers, they do not investigate the co-occurrence of smells on the perspective of lineage.

Future Direction: We suggest that lineage of bad smells is an interesting topic of research, in particular to discover patterns between related bad smells (e.g., FEATURE ENVY and SHOTGUN SURGERY) or smells that co-occur in the same entity. Given the challenge of defining a background for smell lineage, the OBSG community would benefit from an interaction with DCG community, e.g., to reuse the public frameworks already developed (e.g., *SPCP-Miner*[27]).

Wit et al. [102] proposes a plugin that can detect and track clones based on clipboard activities of programmers. This technique is based on the assumption that programmers' copy-paste activities are the primary reason for the creation of code clones and they use this strategy because it is the simplest form of reuse mechanism [210]. Bavota et al. [308] define the smell FEATURE ENVY as *"a method is more interested in a class other than the one it is actually in"*. Additionally, Ahmed et al. [288] reports that the developers tend to perform more frequent copy-paste inside the same file (class). It is reasonable to assume that copy-paste performed across different entities (e.g., between methods on different classes), could introduce the smell FEATURE ENVY. This suggests that the activities inside the editor (e.g., copy-paste) can be monitored to detect and/or prevent the introduction of other bad smells.

10.6 On the new contexts for bad smells

The number of studied smells have been increasing over the last years. We found 104 different smells in the literature and considered several others to be similar, e.g., BLOB/LARGE CLASS. Many of these smells are small variations on each other, and others can be the re-characterization of other smells in specific contexts. For example, considering the runtime environment of the application (*environmental context*), Hecht et al. [368] propose the smell INTERNAL GETTER/SETTER, because in Android the usage of getter/setter is converted into a virtual invocation, which makes the operation three times slower than a direct access. Distefano and Filipović [63] describe some consequences of *memory leaks*. One of these, *Limbo*, can arise from useless objects that occupy a huge amount of memory and are referenced by a long running method: although the objects are not used, they cannot be collected by the *Garbage Collector* (GC). In Android, when the system is running on low memory, the system calls the method *onLowMemory()*. However, if this method is not implemented, the Android system kills the process in order to free the memory [368]. So, in terms of *Environmental Context*, in Android systems the *Limbo pattern* could be seen as a bad smell which can cause an abnormal termination of programs. A variation of this pattern can occur when a "LONG METHOD" (e.g., in terms of LOC) create a list of objects, then performs many operations not related to that list, and only at the end of the method, the objects in the list are handled. Thus, the "bad pattern", occurs because the list is created at the beginning and not at the end. In *Environmental Context*, this pattern can contribute to the situation of low memory and can cause the abnormal termination of the program.

According to Yamashita and Moonen [313], bad smells indicate that there are issues with code quality, such as understandability, e.g., in [327] artifacts with FEATURE ENVY and GOD METHOD were associated with time-consuming changes because they involved highly complex changes in terms of the number of changing points required to complete the task and also regarding the number of elements to be considered simultaneously to complete the task. So, considering the cognitive requirement and the complexity of tasks on artifacts with smells (e.g, FEATURE ENVY and GOD METHOD) and the requirements on performance (e.g., in a swapless Linux or in an embedded environment, when the memory is used fully up, the system kills any arbitrary process to reclaim memory[28]), we observe that relationships with semantical sense were not explicitly studied in the

27. https://homepage.usask.ca/~mam815/spcpminer/

28. http://dl.acm.org/citation.cfm?id=1982324

literature, e.g., are artifacts with co-occurrence of classical smells (e.g., FEATURE ENVY, GOD METHOD) more associated to memory leaks or abnormal terminations? If so, this association could be more intense in languages without *Garbage Collector* (e.g., C++) because memory allocation is managed by the system source code (e.g., *malloc/free*) and, due to the cognitive requirement of an entity with smells, programmers could insert memory leaks.

Future Direction: We suggest that the study of bad smells in *Environmental Context* would reveal new findings/patterns that would have practical outcomes, such as the LIMBO smell.

10.7 Existing taxonomies and their limitations

Taxonomy is defined as "a system for naming and organizing things into groups which share similar qualities[29]". Our systematic literature review reports a large number of smells (104) and they cover many aspects of the software development (e.g., software maintainability, software architecture). Besides the seminal work by Fowler and Beck [9] and Brown et al. [10], other papers also propose taxonomies of smells.

According to Mäntylä et al. [64], a taxonomy makes the smells more understandable and recognizes the relationships between smells. They propose a higher level taxonomy for 22 bad smells identified by Fowler and Beck [9]. Their taxonomy defines seven classes, namely: *bloaters*, *object-orientation abusers*, *change preventers*, *dispensables*, *encapsulators*, *couplers* and *others*. Their findings indicate that a taxonomy for smells could help to explain the correlations between them (e.g., the class *couplers* have a correlation with the class *change preventers*). However, this taxonomy is restricted only to the code smells, and also does not consider the interpretation of subjective definitions of some smells, as example, LARGE CLASS can be interpreted using two points of view (Size — LARGE CLASS ONLY [311] and Complexity — COMPLEX CLASS [317]). Another restriction is related to the way in which programmers think about smells: this taxonomy is not based upon developer intent or on their activities (e.g., refactoring). In this direction, Chatterji et al. [268] conducted a survey with developers and they found many limitations to the current taxonomy of code clones, e.g., taxonomy based on different techniques used in clone detection that helps to determine corresponding techniques and types (which kind of clone is detected by which technique or tool).

Moha et al. [366] proposed a taxonomy that describes the structural relationships among code and design smells, and their measurable, structural, and lexical properties. It also describes the structural relationships among design smells and some code smells. It provides an overview of all key concepts that characterise a design smell and it also makes explicit the relationships among code and design smells. However, in terms of code smells, they use only 17 smells and most of them (65%) are defined by Fowler and Beck [9]. They classify only four design smells (FUNCTIONAL DECOMPOSITION, SPAGHETTI CODE, SWISS ARMY KNIFE and BLOB CLASS).

Wake [65] also proposed classifications of code smells. Their taxonomy distinguished code smells that occur in or among classes. He further distinguished measurable smells, smells related to code duplication, smells due to conditional logic, and others. Nevertheless, this taxonomy also has similar limitations than the previous ones (e.g., small set of smells).

According to Usman et al. [66], the design of a new taxonomy follows six steps: 1) define the subject matter and adopt a definition; 2) specify the descriptive terms that can be used to describe and differentiate subject matter instances; 3) design a classification procedure (qualitative and/or quantitative), which the subject matter instances are systematically assigned to classes or categories; 4) propose a classification structure (e.g., hierarchy) and 5) conduct an validation process (e.g. utility demonstration). In context of sub-optimal code, as previously mentioned, the definitions do not converge (e.g, bad smell, code smell, anti-pattern). In addition, in the last years new types of smells, besides the ones defined by Fowler and Beck [9] and by Brown et al. [10] have emerged, such as smells based on lexicon [43] or smells defined in the context of mobile applications [330, 411], as discussed in the previous subsection. To create a taxonomy of smells, these definitions should be enumerated, normalized (e.g., homogenize words) and thoroughly analyzed, following the above protocol steps, which is not a trivial task.

Future Direction: In general, all taxonomies are restricted to a small set of smells. Thus, new taxonomies that incorporate existing knowledge of well-known bad smells and organize the recent proposed myriad of smells are required to avoid re-inventing specific ad-hoc smell definitions where specialization of more general concepts could facilitate the dissemination of knowledge.

11 CONCLUSION

We conducted a systematic literature review in order to investigate the current relevant body of knowledge about the bad smells. Our review is composed of a set of 351 papers (*Final Database*), which are classified as *Duplicate Code Group* (DCG) or *Other Bad Smell Group* (OBSG). We mainly focused on OBSG papers, because we noticed that DUPLICATED CODE is a very well defined topic with previous surveys already available and a community largely separated from the OBSG community. In addition, DUPLICATED CODE is generally studied alone, while we were interested in understanding how different types of smells are studied together.

We presented the main findings in five *Thematic Areas* (TAs).

TA1: Which. The data revealed that the DUPLICATE CODE bad smell is present in 69.8% of the selected papers, and studies considering the combination of DUPLICATE CODE with other bad smells are not common (7.4% — 18 out of 245). However, analyzing the other kinds of bad smells (OBSG), we observed that they are rather studied together with others (e.g., LONG METHOD occurs in 47.1% of papers that study the smell LARGE CLASS). Currently, most OBSG papers are concentrated in studying bad smells characterized by volume metrics. Moreover, although the number of different kinds of bad smells have been proliferating over the years, the most studied smells are still a half-dozen, mostly related to size and cohesion. It is not

29. Cambridge Dictionaries online, http://dictionary.cambridge.org.

clear that the long tail of less studied kinds of smells can play an important role in assessing design because of their low prevalence, but still they would possibly provide more precise information than just saying that a class is large, without more precise clues on what to do next. We also observed that most of the earliest bad smells (e.g., Brown et al. [10]) have not received due attention, that is, they are peripheral concerning the number of studies. The bad smells by Brown et al. [10] are present in 21.7% of OBSG papers. Similarly, but to a greater extent, the bad smells by Fowler and Beck [9] are present in 82.2% of OBSG papers. From the point of view of the variety of different identified bad smells (104 in total), we observed that most of them (58%) were sporadically studied. In this case, they were studied by up to five papers (see Table 15), that is, they seem to have still just a marginal interest.

TA2: When. Analyzing the studies over time, we observed that studies on DUPLICATE CODE had increased until 2014, but decreased since 2015, whereas studies on the other smells have a linear growth, with a peak in 2013. We also observed that before the year 2011, only 21% of papers studied other types of bad smells and after this period, this value grew up to 45%. This increment of OBSG papers is reflected in the number of authors: between 2010 and 2017 the cumulative number of distinct authors increased of 67%. Thus, the identification and analysis of bad smells is a growing topic that has been continuously attracting interest. The data indicate that the interest (number of papers) in this research topic has been rising over the years, and new researchers are joining the area every year.

TA3: What. Concerning methodological aspects and research findings, the literature presents several types of bad smells and sometimes a given bad smell presents multiple definitions or interpretations. Moreover, different elements of experimental setting such as research aims, subject systems, used tools have large variability. So, the lack of adoption of open science (represented by the insufficient availability of standard and open benchmarks and tools) partially contributes to the existence of multiple ways of analyzing the bad smell impact, thus, explaining why the results of some papers may be contradictory. We have systematically organized the findings produced over more than a decade to unveil the convergent, divergent and main findings. Our results also suggest that studying a particular combination of bad smells just because they are present in the code or because they are conveniently detected by tools may not provide a formal explanation that relates those bad smells (e.g., REFUSED BEQUEST and LONG METHOD). Additionally, co-studying bad smells, i.e., investigating the interaction of different smells on affected code components, looks promising and may help to reveal conditions where bad smells are relevant to software maintenance. We also observed that 21% of papers classified as co-occurrence can be actually considered as co-studies and only 18% of the 104 smells documented in our dataset are investigated in empirical co-studies. As mentioned previously and also reported by Zhang et al. [7], many bad smells have a marginal interest (e.g., MIDDLE MAN, PRIMITIVE OBSESSION, PARALLEL INHERITANCE HIERARCHIES). We think that they have been neglected because there is few evidences of their negative impact on source code. However, it would be interesting to include these smells in co-studies investigating their interaction with widely studied bad smells. For example, Mäntylä et al. [64] report that PARALLEL INHERITANCE HIERARCHIES cause REFUSED BEQUEST, which in turn may be related to LONG METHODS. Finally, we also identified the main purposes of papers and our data shows that *Detection* (30.7%), *Impact* (24.3%), and *Qualitative Characterization* (11.4%) are the most frequent aims.

TA4: Who. Concerning the researchers studying other bad smells than DUPLICATE CODE, we observed that they have several levels of interest in the subject, that is, some of them publish sporadically and others continuously. There is a large connected graph of collaborating authors that has the *small world* property, indicating high flow of information within that group of researchers. On the other hand, there is a large number of smaller disconnected graphs which still produce a significant part of studies (44%). This division between a large connected group and many isolated groups unveils the dynamics of how the findings are being produced. We also observed that some research centers are highly interconnected to others.

TA5: Where. We showed that some venues have a much higher proportion of studies on DUPLICATE CODE, while others are more balanced with other bad smells. The presence in the list of topics of some conference of specific topics concerned with DUPLICATED CODE and generic topics on bad smells might explain why the community working only on DUPLICATE CODE and the community working on the other types of smells are largely separated.

In general, our results show that a large body of knowledge has been produced on bad smells. Nonetheless, there are still some divergences due to the lack of generality of current studies. Moreover, despite the several different kinds of associations that have been found regarding bad smells in code, there is little evidence on cause-effect relations that would provide more actionable knowledge on available empirical data.

ACKNOWLEDGMENTS

The authors would like to thank the partial funding of the Brazilian governmental agencies FAPEMIG, CNPq and CAPES.

TABLE 14. Tools/techniques used by the OBSG papers to handle the smells.

	Large Class (Blob class, God Class)	Feature Envy	Long Method (God Method)	Data Class	Shotgun Surgery	Refused Bequest	Long Parameter List	Spaghetti Code	Duplicated Code	Message Chains Class	Function Class (Func. Decomposition)	Abstract Class (Speculative Generality)	Few Methods (Lazy classes, Small Class)	Data Clumps	Swiss Army Knife	Complex Class Only	Field Public (CDSBP)	Divergent Change	Misplaced Class	Brain Method	Temporary variable, several purposes	Intensive Coupling	Dispersed (Extensive) Coupling	Switch Statements	Unit Test Smells	AntiSingleton
[294]		X																								
2D-DSL	X		X			X	X	X			X	X	X		X											
Noseprints	X	X	X	X		X	X			X				X												
Prodetection	X	X	X	X	X	X													X							
P-EA	X	X		X	X		X	X			X		X													
BLOP	X	X		X			X	X			X		X													
HistoryMiner	X							X								X	X								X	
BBT	X			X	X															X			X			
HIST	X	X			X													X								
[295]		X			X															X		X	X			
TACO	X	X	X																X							
BDTEX	X							X			X				X											
[296]																										
[297]	X			X		X																				
[298]		X	X						X																	
SMURF	X							X			X				X											
[299]			X				X					X														
[300]	X							X			X															
[301]					X				X									X								
[302]	X							X			X															
[303]	X							X			X															
EvolutionAnalyzer	X																									
Fluid Tool		X		X																						
[304]			X						X																	
[305]	X			X																						
[306]	X				X																					
[307]																										
Van	X			X																						
AntiPattern Scanner				X																						
Methodbook		X																								
OBEY																										
Teamscale			X																							
TestEvoHound																									X	
Borland Together	X	X	X	X	X	X	X		X	X		X	X	X				X	X		X			X		
Understand	X	X	X	X	X		X						X					X	X							
Pascal Analyzer	X						X																			
SonarQube									X																	
inFusion/Incode	X	X		X	X	X	X		X	X		X	X	X				X	X	X	X	X	X	X		
SCOOP/Organic	X	X	X	X	X		X											X								
CodeVizard	X				X																					
IYC	X		X																							
TestSmells																									X	
DECOR	X	X	X	X	X	X	X	X		X	X	X	X		X	X	X									X
MuLATo	X	X	X	X	X		X						X					X	X							
iPlasma	X	X	X	X	X	X	X		X			X	X							X		X	X			
[308]	X	X	X			X	X	X		X		X	X			X	X									
Stench Blossom	X	X	X						X	X				X										X		
SpIRIT	X	X		X	X	X														X		X	X			
jDeodorant	X	X	X				X		X									X						X		
InsRefactor	X		X	X			X		X								X							X		
PMD	X		X				X		X							X							X			
JCodeOdor	X			X	X					X										X			X			
JSNOSE	X						X						X											X		
Cbsdetector										X		X		X										X		
HULK	X			X											X											
Checkstyle	X		X				X		X																	
[309]	X	X	X	X																						
[310]		X	X						X																	
Paprika	X		X													X										
Metrics	X		X				X																			
SourceMiner	X	X																X								
ConQAT			X						X																	
SDMetrics	X						X																			

	Tradition Breaker	Large Class Only	Interface Segregation Principle Violation	Schizophrenic class	Brain Class	Duplicated code in conditional branches	Use interface instead of implementation	Middle Man	Parallel Inheritance Hierarchies	Inappropriate Intimacy	Ambiguous Interface	Primitive Obsession	Lexicon Bad Smells	Type Check (State Check)	Duplicate Pointcut	God Pointcut	Redundant Pointcut	Connector Envy	Component Concern Overload	Scattered Parasitic Functionality	Smells in Android (Specific)	Controller Class	Low Cohesion Only	Classes with Different Interfaces	Wide Subsystem Interface	Instanceof
[294]															X	X	X									
2D-DSL		X																								
Noseprints								X				X														
Prodetection																									X	
P-EA																										
BLOP																										
HistoryMiner																				X						
BBT																										
HIST									X																	
[295]																										
TACO																										
BDTEX																										
[296]												X														
[297]				X																						
[298]													X													
SMURF																										
[299]																										
[300]																										
[301]																										
[302]																										
[303]																										
EvolutionAnalyzer					X																					
Fluid Tool																										
[304]																										
[305]																										
[306]																										
[307]											X															
Van																										
AntiPattern Scanner																										
Methodbook																										
OBEY								X																		
Teamscale																										
TestEvoHound																										
Borland Together		X	X			X	X			X	X													X	X	
Understand		X								X	X				X	X	X	X	X	X						
Pascal Analyzer																										
SonarQube																										
inFusion/Incode	X		X	X			X																	X	X	
SCOOP/Organic											X							X	X	X						
CodeVizard																										
IYC																										
TestSmells																										
DECOR	X	X						X		X												X	X			
MuLATo		X								X	X				X	X	X	X	X	X						
iPlasma	X			X	X																					
[308]																										
Stench Blossom												X		X												X
SpIRIT	X				X																					
jDeodorant														X												
InsRefactor																										
PMD																										
JCodeOdor																										
JSNOSE																										
Cbsdetector								X																		
HULK																						X	X			
Checkstyle																										
[309]																										
[310]										X																
Paprika																					X					
Metrics																										
SourceMiner																										
ConQAT																										
SDMetrics																										

	Typecasts	God Aspect	Composition Bloat	Forced Join Point	Lazy Aspect	Linguistic Antipatterns	Anonymous Pointcut	Lava Flow (dead code)	Extraneous Connector	Cyclic Dependency	Idle Pointcut	Child Class	Class Global Variable	Class One Method	Field Private	Has Children	Many Attributes	Method No Parameter	Multiple Interface	No Inheritance	No Polymorphism	Not Abstract	Not Complex	One Child Class	Parent Class Provides Protected	Rare Overriding
[294]					X		X				X															
2D-DSL																										
Noseprints																										
Prodetection																										
P-EA																										
BLOP																										
HistoryMiner																										
BBT																										
HIST																										
[295]																										
TACO																										
BDTEX																										
[296]																										
[297]																										
[298]																										
SMURF																										
[299]																										
[300]																										
[301]																										
[302]																										
[303]																										
EvolutionAnalyzer																										
Fluid Tool																										
[304]																										
[305]																										
[306]																										
[307]																										
Van																										
AntiPattern Scanner																										
Methodbook																										
OBEY																										
Teamscale																										
TestEvoHound																										
Borland Together																										
Understand		X	X	X					X																	
Pascal Analyzer																										
SonarQube																										
inFusion/Incode										X																
SCOOP/Organic									X	X																
CodeVizard																										
IYC																										
TestSmells																										
DECOR												X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
MuLATo		X	X	X					X																	
iPlasma																										
[308]																										
Stench Blossom	X																									
SpIRIT																										
jDeodorant																										
InsRefactor																										
PMD								X																		
JCodeOdor																										
JSNOSE																										
Cbsdetector																										
HULK																										
Checkstyle																										
[309]																										
[310]																										
Paprika																										
Metrics																										
SourceMiner																										
ConQAT																										
SDMetrics																										

	Two Inheritance	Incomplete Library Class	Comments	Simulation of multiple inheritance	Useless Field	Useless Method	Useless Class	Empty catch blocks	BaseClassKnowsDerivedClass	BaseClassShouldBeAbstract	ManyFieldAttributesButNotComplex	Junk Material	Borrowed Pointcut	Various Concerns	Abstract Method Introduction	God Package	Extraneous Adjacent Connector	Unused Interface	Promiscuous Package	Distorted Hierarchy	Smells in MVC Arq. (Specific)	Smells in CSS (Specific - DSL)	Smells in JavaScript (Specific - DSL)	Obsolete Parameter	Annotation Bundle	Smells in Puppet (Specific - DSL)
[294]												X	X	X	X											
2D-DSL																										
Noseprints																										
Prodetection																X										
P-EA																										
BLOP																										
HistoryMiner																										
BBT																										
HIST																										
[295]																										
TACO																			X							
BDTEX																										
[296]					X	X	X																			
[297]																										
[298]																										
SMURF																										
[299]																										
[300]																										
[301]																										
[302]																										
[303]																										
EvolutionAnalyzer																										
Fluid Tool																										
[304]																										
[305]																										
[306]																										
[307]																								X		
Van																										
AntiPattern Scanner																										
Methodbook																										
OBEY																										
Teamscale																										
TestEvoHound																										
Borland Together				X																						
Understand																										
Pascal Analyzer																										
SonarQube																										
inFusion/Incode				X																X						
SCOOP/Organic																		X								
CodeVizard																										
IYC																										
TestSmells																										
DECOR	X								X	X	X															
MuLATo																										
iPlasma																										
[308]																										
Stench Blossom			X																							
SpIRIT																										
jDeodorant																										
InsRefactor																										
PMD																										
JCodeOdor																										
JSNOSE								X															X			
Cbsdetector																										
HULK																										
Checkstyle																										
[309]																										
[310]																										
Paprika																										
Metrics																										
SourceMiner																										
ConQAT																										
SDMetrics																										

continued on next page

continued from previous page

	Large Class (Blob class, God Class)	Feature Envy	Long Method (God Method)	Data Class	Shotgun Surgery	Refused Bequest	Long Parameter List	Spaghetti Code	Duplicated Code	Message Chains Class	Function Class (Func. Decomposition)	Abstract Class (Speculative Generality)	Few Methods (Lazy classes, Small Class)	Data Clumps	Swiss Army Knife	Complex Class Only	Field Public (CDSBP)	Divergent Change	Misplaced Class	Brain Method	Temporary variable, several purposes	Intensive Coupling	Dispersed (Extensive) Coupling	Switch Statements	Unit Test Smells	AntiSingleton
Hot-Pepper																										
JCCD									X																	
JMove		X																								
LAPD																										
LBSDetectors																										
NiCad									X																	
Puppeteer																										
SAME									X																	
SKUNK																										
SmellCSS																										
SmellJS																										
SmellyCat																										
TestHound																									X	
Trex																									X	

	Tradition Breaker	Large Class Only	Interface Segregation Principle Violation	Schizophrenic class	Brain Class	Duplicated code in conditional branches	Use interface instead of implementation	Middle Man	Parallel Inheritance Hierarchies	Inappropriate Intimacy	Ambiguous Interface	Primitive Obsession	Lexicon Bad Smells	Type Check (State Check)	Duplicate Pointcut	God Pointcut	Redundant Pointcut	Connector Envy	Component Concern Overload	Scattered Parasitic Functionality	Smells in Android (Specific)	Controller Class	Low Cohesion Only	Classes with Different Interfaces	Wide Subsystem Interface	Instanceof
Hot-Pepper																					X					
JCCD																										
JMove																										
LAPD																										
LBSDetectors													X													
NiCad																										
Puppeteer																										
SAME																										
SKUNK																										
SmellCSS																										
SmellJS																										
SmellyCat																										
TestHound																										
Trex																										

	Typecasts	God Aspect	Composition Bloat	Forced Join Point	Lazy Aspect	Linguistic Antipatterns	Anonymous Pointcut	Lava Flow (dead code)	Extraneous Connector	Cyclic Dependency	Idle Pointcut	Child Class	Class Global Variable	Class One Method	Field Private	Has Children	Many Attributes	Method No Parameter	Multiple Interface	No Inheritance	No Polymorphism	Not Abstract	Not Complex	One Child Class	Parent Class Provides Protected	Rare Overriding
Hot-Pepper																										
JCCD																										
JMove																										
LAPD						X																				
LBSDetectors																										
NiCad																										
Puppeteer																										
SAME																										
SKUNK																										
SmellCSS																										
SmellJS																										
SmellyCat																										
TestHound																										
Trex																										

	Two Inheritance	Incomplete Library Class	Comments	Simulation of multiple inheritance	Useless Field	Useless Method	Useless Class	Empty catch blocks	BaseClassKnowsDerivedClass	BaseClassShouldBeAbstract	ManyFieldAttributesButNotComplex	Junk Material	Borrowed Pointcut	Various Concerns	Abstract Method Introduction	God Package	Extraneous Adjacent Connector	Unused Interface	Promiscuous Package	Distorted Hierarchy	Smells in MVC Arq. (Specific)	Smells in CSS (Specific - DSL)	Smells in JavaScript (Specific - DSL)	Obsolete Parameter	Annotation Bundle	Smells in Puppet (Specific - DSL)
Hot-Pepper																										
JCCD																										
JMove																										
LAPD																										
LBSDetectors																										
NiCad																										
Puppeteer																										X
SAME																										
SKUNK																									X	
SmellCSS																						X				
SmellJS																							X			
SmellyCat																					X					
TestHound																										
Trex																										

Ad-Hoc	Commercial	Deprecated	Public

TABLE 15. Bad smells of each paper in OBSG.

Paper ID	Large Class (Blob class, God Class)	Feature Envy	Long Method (God Method)	Data Class	Shotgun Surgery	Refused Bequest	Long Parameter List	Spaghetti Code	Duplicated Code	Message Chains Class	Few Methods (Lazy Class, Small Class)	Abstract Class (Speculative Generality)	Function Class (Func. Decomposition)	Data Clumps	Complex Class Only	Swiss Army Knife	Field Public (CDSBP)	Divergent Change	Misplaced Class	Temporary Variable, Several Purposes	Brain Method	Intensive Coupling	Dispersed (Extensive) Coupling	Interface Segregation Principle Violation	Switch Statements	Unit Test Smells	AntiSingleton	Tradition Breaker	Large Class Only	Duplicated Code in Conditional Branches	Use Interface Instead of Implementation	Schizophrenic Class	Brain Class	Middle Man	Parallel Inheritance Hierarchies	Inappropriate Intimacy	Ambiguous Interface	Primitive Obsession	Lexicon Bad Smells	Type Check (State Check)	Duplicate Pointcut	God Pointcut	Redundant Pointcut	Connector Envy	Component Concern Overload	Scattered Parasitic Functionality	Smells in Android (Specific)	Controller Class	Low Cohesion Only	Classes with Different Interfaces	Wide Subsystem Interface	Instanceof	Typecasts	God Aspect	Composition Bloat	Forced Join Point	Lazy Aspect	Linguistic Antipatterns	Anonymous Pointcut	Lava Flow (Dead Code)	Extraneous Connector	Cyclic Dependency	Idle Pointcut	Child Class	Class Global Variable	Class One Method	Field Private	Has Children	Many Attributes	Method No Parameter	Multiple Interface	No Inheritance	No Polymorphism	Not Abstract	Not Complex	One Child Class	Parent Class Provides Protected	Rare Overriding	Two Inheritance	Incomplete Library Class	Comments	Simulation of Multiple Inheritance	Useless Field	Useless Method	Useless Class	Empty Catch Blocks	BaseClassKnowsDerivedClass	BaseClassShouldBeAbstract	ManyFieldAttributesButNotComplex	Junk Material	Borrowed Pointcut	Various Concerns	Abstract Method Introduction	God Package	Extraneous Adjacent Connector	Unused Interface	Promiscuous Package	Distorted Hierarchy	Smells in MVC Arq. (Specific)	Smells in CSS (Specific - [1]DSL)	Smells in JavaScript (Specific - [1]DSL)	Obsolete Parameter	Annotation Bundle	Smells in Puppet (Specific - [1]DSL)	#
[311]○	X		X	X			X			X	X	X	X		X		X												X																				X	X														X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X																										29
[312]○	X	X	X	X	X	X	X		X	X	X	X		X				X		X					X									X	X	X		X												X										X																				X	X																							23
[313]○	X	X	X		X	X	X		X	X	X	X		X				X	X	X				X	X																									X	X																																	X																						19
[314]○	X		X			X	X	X		X	X	X	X		X	X	X										X	X																																																												X	X	X																17
[315]●	X	X	X	X	X		X				X							X	X										X												X	X	X											X	X	X																																																								16
[299]○	X	X	X	X	X	X	X		X			X									X	X	X					X				X	X																																																																										15
[316]●																																				X				X	X	X	X	X	X								X	X	X	X		X				X																																									X													14
[317]○	X		X			X	X	X		X	X	X			X	X	X										X		X																																																																																13
[318]○	X	X	X		X		X											X	X																	X	X							X	X	X															X																																																								13
[319]●	X		X			X	X	X		X	X	X			X	X	X										X		X																																																																															13
[320]●	X	X	X	X	X													X																		X							X	X	X															X	X																																													X										13
[321]○	X	X		X	X	X			X	X				X							X	X						X				X																														X																																																								13
[322]○	X	X	X	X	X	X								X					X	X				X						X	X																																																																																				12
[323]○	X	X	X	X	X	X								X					X	X				X						X	X																																																																																						12
[324]●	X	X	X	X	X	X								X					X	X				X						X	X																																																																																							12
[325]○	X		X			X	X	X		X	X	X			X	X	X										X																																																																																			12
[326]○	X	X	X			X	X	X			X	X			X		X																	X		X																																																																													12
[327]●	X	X	X	X	X	X								X					X	X				X						X	X																																																																																					12
[328]●	X	X	X	X	X	X								X					X	X				X						X	X																																																																																							12
[329]●	X	X		X	X	X			X	X				X								X						X				X																																																																												X							12
[330]○	X	X	X	X	X	X								X					X	X				X						X	X																																																																																						12
[331]○	X		X			X	X	X		X		X			X	X	X										X																																																																																			11
[294]●		X																																							X	X	X														X		X				X																											X	X	X	X												11
[332]○	X		X			X	X	X		X		X			X	X	X										X																																																																																			11
[308]○	X	X	X			X	X	X		X	X	X			X		X																																																																																													11
[333]○	X	X			X	X		X	X				X			X		X																	X					X																																																																													11
[334]○	X		X			X	X			X		X			X	X	X										X																																																																																				10
[335]●	X		X			X	X	X			X	X	X			X													X																																																																																			10
[336]○	X	X	X	X		X	X			X				X																				X				X																																																																															10
[337]○	X	X		X	X	X															X	X	X					X					X																																																																																						10
[338]○	X	X		X	X	X															X	X	X					X					X																																																																																						10
[339]○	X	X	X	X	X	X													X																																X																																														X											9
[296]●	X	X	X				X		X																													X																																																		X	X	X																				9
[340]○	X	X		X	X		X	X			X		X																																																																																																							8
[341]○	X			X		X			X					X							X	X										X																																																																																				8
[342]○	X	X	X							X				X											X																											X	X																																																															8
[343]○	X		X	X			X		X								X								X																																																																																										7
[344]○	X	X		X					X					X														X				X																																																																																				7
[345]○	X	X	X		X														X			X	X																																																																																															7
[346]●	X	X	X	X	X		X											X																																																																																																			7
[347]○	X	X		X			X	X			X		X																																																																																																							7
[348]○	X	X	X	X	X	X																																																																																																																			6
[349]●	X	X	X	X	X													X																																																																																																					6
[350]○	X	X	X				X		X																				X																																																																																								6
[351]●	X							X					X		X		X									X																																																																																										6
[352]○	X			X	X					X											X		X																																																																																															6
[353]○	X			X	X					X											X		X																																																																																															6
[354]○	X	X			X													X																	X																																																																																					5
[355]○	X						X				X														X																																																															X																							5
[356]○	X			X									X			X																																												X																																																							5
[357]●										X		X		X											X									X																																																																																							5
[358]○	X			X												X																																X	X																																																																					5
[359]○	X	X	X																X																																																																																X									5
[360]○	X							X					X		X		X																																																																																																			5
[361]○	X	X			X													X																	X																																																																																					5
[306]○	X	X	X		X	X																																																																																																																		5
[295]○		X			X																X	X	X																																																																																															5
[362]○	X			X	X																X		X																																																																																																5
[363]○	X						X	X			X	X																																																																																																										5
[364]○	X							X					X			X																																																																																																					4
[365]●	X							X					X			X																																																																																																					4
[366]●	X							X					X			X																																																																																																					4

continued on next page

continued from previous page

Paper ID	Large Class (Blob class, God Class)	Feature Envy	Long Method (God Method)	Data Class	Shotgun Surgery	Refused Bequest	Long Parameter List	Spaghetti Code	Duplicated Code	Message Chains Class	Few Methods (Lazy Class, Small Class)	Abstract Class (Speculative Generality)	Function Class (Func. Decomposition)	Data Clumps	Complex Class Only	Swiss Army Knife	Field Public (CDSBP)	Divergent Change	Misplaced Class	Temporary Variable, Several Purposes	Brain Method	Intensive Coupling	Dispersed (Extensive) Coupling	Interface Segregation Principle Violation	Switch Statements	Unit Test Smells
[367]○	X	X		X																						
[309]○	X	X	X	X																						
[368]○	X		X												X											
[297]○	X			X		X																				
[310]○		X	X						X																	
[298]○		X	X						X																	
[369]○	X	X	X																							
[370]○	X		X								X				X											
[371]○	X						X		X																	
[372]○	X		X						X																	
[373]○	X							X					X													
[300]○	X							X					X													
[374]○		X	X																							
[375]●	X	X		X																						
[376]○		X	X																						X	
[301]○					X				X									X								
[377]○	X	X																X								
[302]○	X							X					X													
[303]○	X							X					X													
[378]○	X																									
[379]○																										
[380]○	X				X																					
[381]○			X						X																	
[382]●	X							X																		
[383]○	X			X																						
[384]○											X									X						
[304]○			X						X																	
[385]○	X			X																						
[305]○	X			X																						
[307]○																										
[386]○	X		X																							
[387]	X																									
[388]																										
[389]	X																									
[390]	X																									
[391]		X																								
[392]		X																								
[393]		X																								
[394]																										X
[395]																										
[396]	X																									
[397]																										X
[398]																										
[399]																										
[400]	X																									
[401]																										
[402]																										X
[403]	X																									
[404]	X																									
[405]																										X
[406]																										X
[407]																										X
[408]	X																									
[409]																										
[410]																										
[411]																										
[412]																										
[413]																										
[414]			X																							
[415]																										
[416]																										
[417]																										
	87	49	48	37	32	30	26	23	18	18	17	16	16	15	14	14	13	12	12	9	9	8	8	7	7	7

Paper ID	AntiSingleton	Tradition Breaker	Large Class Only	Duplicated Code in Conditional Branches	Use Interface Instead of Implementation	Schizophrenic Class	Brain Class	Middle Man	Parallel Inheritance Hierarchies	Inappropriate Intimacy	Ambiguous Interface	Primitive Obsession	Lexicon Bad Smells	Type Check (State Check)	Duplicate Pointcut	God Pointcut	Redundant Pointcut	Connector Envy	Component Concern Overload	Scattered Parasitic Functionality	Smells in Android (Specific)	Controller Class	Low Cohesion Only	Classes with Different Interfaces	Wide Subsystem Interface	Instanceof
[367]○							X																			
[309]○																										
[368]○																					X					
[297]○						X																				
[310]○										X																
[298]○													X													
[369]○														X												
[370]○																										
[371]○																										
[372]○																										
[373]○																										
[300]○																										
[374]○														X												
[375]●																										
[376]○																										
[301]○																										
[377]○																										
[302]○																										
[303]○																										
[378]○							X																			
[379]○																										X
[380]○																										
[381]○																										
[382]●																										
[383]○																										
[384]○																										
[304]○																										
[385]○																										
[305]○																										
[307]○											X															
[386]○																										
[387]																										
[388]								X																		
[389]																										
[390]																										
[391]																										
[392]																										
[393]																										
[394]																										
[395]													X													
[396]																										
[397]																										
[398]																										
[399]													X													
[400]																										
[401]																										
[402]																										
[403]																										
[404]																										
[405]																										
[406]																										
[407]																										
[408]																										
[409]																										
[410]																										
[411]																					X					
[412]																										
[413]																										
[414]																										
[415]																										
[416]																					X					
[417]																										
	7	7	6	6	6	6	5	5	4	4	4	3	3	3	3	3	3	3	3	3	3	2	2	2	2	2

Paper ID	Typecasts	God Aspect	Composition Bloat	Forced Join Point	Lazy Aspect	Linguistic Antipatterns	Anonymous Pointcut	Lava Flow (Dead Code)	Extraneous Connector	Cyclic Dependency	Idle Pointcut	Child Class	Class Global Variable	Class One Method	Field Private	Has Children	Many Attributes	Method No Parameter	Multiple Interface	No Inheritance	No Polymorphism	Not Abstract	Not Complex	One Child Class	Parent Class Provides Protected	Rare Overriding
[367]○																										
[309]○																										
[368]○																										
[297]○																										
[310]○																										
[298]○																										
[369]○																										
[370]○																										
[371]○																										
[372]○																										
[373]○																										
[300]○																										
[374]○																										
[375]●																										
[376]○																										
[301]○																										
[377]○																										
[302]○																										
[303]○																										
[378]○																										
[379]○	X																									
[380]○																										
[381]○																										
[382]●																										
[383]○																										
[384]○																										
[304]○																										
[385]○																										
[305]○																										
[307]○																										
[386]○																										
[387]																										
[388]																										
[389]																										
[390]																										
[391]																										
[392]																										
[393]																										
[394]																										
[395]																										
[396]																										
[397]																										
[398]						X																				
[399]																										
[400]																										
[401]						X																				
[402]																										
[403]																										
[404]																										
[405]																										
[406]																										
[407]																										
[408]																										
[409]																										
[410]																										
[411]																										
[412]																										
[413]																										
[414]																										
[415]																										
[416]																										
[417]																										
	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Paper ID	Two Inheritance	Incomplete Library Class	Comments	Simulation of Multiple Inheritance	Useless Field	Useless Method	Useless Class	Empty Catch Blocks	BaseClassKnowsDerivedClass	BaseClassShouldBeAbstract	ManyFieldAttributesButNotComplex	Junk Material	Borrowed Pointcut	Various Concerns	Abstract Method Introduction	God Package	Extraneous Adjacent Connector	Unused Interface	Promiscuous Package	Distorted Hierarchy	Smells in MVC Arq. (Specific)	Smells in CSS (Specific - [1]DSL)	Smells in JavaScript (Specific - [1]DSL)	Obsolete Parameter	Annotation Bundle	Smells in Puppet (Specific - [1]DSL)	#
[367]○																											4
[309]○																											4
[368]○																											4
[297]○																											4
[310]○																											4
[298]○																											4
[369]○																											4
[370]○																											4
[371]○																											3
[372]○																											3
[373]○																											3
[300]○																											3
[374]○																											3
[375]●																											3
[376]○																											3
[301]○																											3
[377]○																											3
[302]○																											3
[303]○																											3
[378]○																											2
[379]○																											2
[380]○																											2
[381]○																											2
[382]●																											2
[383]○																											2
[384]○																											2
[304]○																											2
[385]○																											2
[305]○																											2
[307]○																								X			2
[386]○																											2
[387]																											1
[388]																											1
[389]																											1
[390]																											1
[391]																											1
[392]																											1
[393]																											1
[394]																											1
[395]																											1
[396]																											1
[397]																											1
[398]																											1
[399]																											1
[400]																											1
[401]																											1
[402]																											1
[403]																											1
[404]																											1
[405]																											1
[406]																											1
[407]																											1
[408]																											1
[409]																					X						1
[410]																						X					1
[411]																											1
[412]																							X				1
[413]																									X		1
[414]																											1
[415]																										X	1
[416]																											1
[417]																											
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	

[1]Domain Specific Language - DSL.
○ Co-occurrence of smells; ● Co-study of smells (it is a subset of co-occurrence).

REFERENCES

[1] X. S. Dexun Jiang, Peijun Ma and T. Wang, "Distance metric based divergent change bad smell detection and refactoring scheme analysis," *International Journal of Innovative Computing, Information and Control*, vol. 10, no. 4, pp. 1519–1531, 2014.

[2] C. Seaman and Y. Guo, "Chapter 2 - measuring and monitoring technical debt," ser. Advances in Computers, M. V. Zelkowitz, Ed. Elsevier, 2011, vol. 82, pp. 25 – 46.

[3] J. V. Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of Systems and Software*, vol. 61, no. 2, pp. 105 – 119, 2002.

[4] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*. Elsevier Science, 2014.

[5] A. Bandi, B. Williams, and E. Allen, "Empirical evidence of code decay: A systematic mapping study," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 341–350.

[6] J. R. Pate, R. Tairas, and N. A. Kraft, "Clone evolution: a systematic review," *Journal of Software: Evolution and Process*, vol. 25, no. 3, pp. 261–283, 2013.

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

[8] R. Oliveto, M. Gethers, G. Bavota, D. Poshyvanyk, and A. De Lucia, "Identifying method friendships to remove the feature envy bad smell," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. New York, New York, USA: ACM Press, 2011, p. 820.

[9] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, ser. Object Technology Series. Addison-Wesley, 1999.

[10] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York, NY, USA: John Wiley & Sons, Inc., 1998.

[11] B. L. Sousa, M. A. S. Bigonha, and K. A. M. Ferreira, "A systematic literature mapping on the relationship between design patterns and bad smells," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18. New York, NY, USA: ACM, 2018, pp. 1528–1535.

[12] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '16. New York, NY, USA: ACM, 2016, pp. 18:1–18:12.

[13] G. Vale, E. Figueiredo, R. Abílio, and H. Costa, "Bad smells in software product lines: A systematic review," in *2014 Eighth Brazilian Symposium on Software Components, Architectures and Reuse*, Sept 2014, pp. 84–94.

[14] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 9–19.

[15] B. A. Kitchenham, D. Budgen, and O. Pearl Brereton, "Using mapping studies as the basis for further research - a participant-observer case study," *Inf. Softw. Technol.*, vol. 53, no. 6, pp. 638–651, Jun. 2011.

[16] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," School of Computer Science andMathematics, Keele University and Department of Computer Science, University of Durham, Tech. Rep. EBSE Technical Report EBSE-2007-01, 2007.

[17] F. Christos and O. D. W., "A survey of information retrieval and filtering methods," Univ. of Maryland Institute for Advanced Computer Studies Report, College Park, MD, USA, Tech. Rep. CS-TR-3514, 1995.

[18] T.-H. Chen, S. W. Thomas, and A. E. Hassan, "A survey on the use of topic models when mining software repositories," *Empirical Software Engineering*, pp. 1–77, 2015.

[19] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall, "Mining Software Evolution to Predict Refactoring," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, 2007, pp. 354–363.

[20] J. Altidor and Y. Smaragdakis, "Refactoring Java Generics by Inferring Wildcards, in Practice," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, pp. 271–290.

[21] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, oct 2011.

[22] G. Bavota, A. Lucia, A. Marcus, and R. Oliveto, "Automating Extract Class Refactoring: An Improved Method and Its Evaluation," *Empirical Softw. Engg.*, vol. 19, no. 6, pp. 1617–1664, 2014.

[23] W. F. Opdyke and R. E. Johnson, "Refactoring: An aid in designing application frameworks and evolving object-oriented systems," in *Proceedings of SOOPPA'90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*. New York, New York, USA: ACM Press, 1990.

[24] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, Champaign, IL, USA, 1992, uMI Order No. GAX93-05645.

[25] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[26] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977.

[27] S. Kimura, Y. Higo, H. Igaki, and S. Kusumoto, "Move code refactoring with dynamic analysis," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, sep 2012, pp. 575–578.

[28] M. Gawade, K. Ravikanth, and S. Aggarwal, "Constantine: configurable static analysis tool in Eclipse," *Software: Practice and Experience*, vol. 44, no. 5, pp. 537–563, 2014.

[29] K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "CRat: A refactoring support tool for Form Template Method," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, jun 2012, pp. 250–252.

[30] R. Mahouachi, M. Kessentini, and K. Ghedira, "A New Design Defects Classification: Marrying Detection and Correction," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, J. de Lara and A. Zisman, Eds. Springer Berlin Heidelberg, 2012, vol. 7212, pp. 455–470.

[31] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Berlin Heidelberg, 2006.

[32] T. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, pp. 308–320, Dec 1976.

[33] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoring challenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, July 2014.

[34] A. V. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, 2001, pp. 92–95.

[35] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object Oriented Reengineering Patterns*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[36] J. Hannemann and G. Kiczales, "Design pattern implementation in java and aspectj," in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '02. New York, NY, USA: ACM, 2002, pp. 161–173.

[37] R. Marinescu, "Measurement and quality in object-oriented design," Ph.D. dissertation, "Politehnica" University of Timisoara, Department of Computer Science, 2002.

[38] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[39] M. P. Monteiro and J. a. M. Fernandes, "Towards a catalog of aspect-oriented refactorings," in *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, ser. AOSD '05. New York, NY, USA: ACM, 2005, pp. 111–122.

[40] E. K. Piveta, M. Hecht, M. S. Pimenta, and R. T. Price, "Detecting bad smells in aspectj," *Journal of Universal Computer Science*, 2006.

[41] K. Srivisut and P. Muenchaisri, "Bad-smell metrics for aspect-oriented software," in *Computer and Information Science, 2007. ICIS 2007. 6th IEEE/ACIS International Conference on*, July 2007, pp. 1060–1065.

[42] A. Trifu and U. Reupke, "Towards automated restructuring of object oriented systems," in *Software Maintenance and Reengineering, 2007. CSMR '07. 11th European Conference on*, March 2007, pp. 39–48.

[43] S. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "Lexicon bad smells in software," in *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, Oct 2009, pp. 95–99.

[44] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying Architectural Bad Smells," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 255–258.

[45] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 609–613.

[46] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[47] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 International Symposium on Empirical Software Engineering and Measurement*, Sept 2011, pp. 275–284.

[48] J. Tian, *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*, ser. Wiley - IEEE. Wiley, 2005.

[49] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 91–100.

[50] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *2011 33rd International Conference on Software Engineering (ICSE)*, May 2011, pp. 1037–1039.

[51] B. F. dos Santos Neto, M. Ribeiro, V. T. da Silva, C. Braga, C. J. P. de Lucena, and E. de Barros Costa, "Autorefactoring: A platform to build refactoring agents," *Expert Systems with Applications*, vol. 42, no. 3, pp. 1652 – 1664, 2015.

[52] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, Sept 2013, pp. 392–395.

[53] N. Moha, A. M. Rouane Hacene, P. Valtchev, and Y.-G. Guéhéneuc, *Formal Concept Analysis: 6th International Conference, ICFCA 2008, Montreal, Canada, February 25-28, 2008. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, ch. Refactorings of Design Defects Using Relational Concept Analysis, pp. 289–304.

[54] I. Macia, R. Arcoverde, E. Cirilo, A. Garcia, and A. von Staa, "Supporting the identification of architecturally-relevant code anomalies," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Sept 2012, pp. 662–665.

[55] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, "Jdeodorant: Clone refactoring," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, May 2016, pp. 613–616.

[56] D. R. Farine, C. J. Garroway, and B. C. Sheldon, "Social network analysis of mixed-species flocks: exploring the structure and evolution of interspecific social behaviour," *Animal Behaviour*, vol. 84, no. 5, pp. 1271 – 1277, 2012.

[57] M. Oliveira and J. Gama, "An overview of social network analysis," *Wiley Interdisciplinary Reviews: Data*

Mining and Knowledge Discovery, vol. 2, no. 2, pp. 99–115, 2012.

[58] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," 2009.

[59] A. Field, *Discovering Statistics Using SPSS*, ser. Introducing Statistical Methods Series. SAGE Publications, 2007.

[60] Q. K. Telesford, K. E. Joyce, S. Hayasaka, J. H. Burdette, and P. J. Laurienti, "The ubiquity of small-world networks," *Brain Connectivity*, vol. 1, no. 5, pp. 367–375, 2011.

[61] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. PP, no. to appear, pp. 1–1, 2017.

[62] F. Palomba, D. D. Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Landfill: An open dataset of code smells with public evaluation," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 482–485.

[63] D. Distefano and I. Filipović, *Memory Leaks Detection in Java by Bi-abductive Inference*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 278–292.

[64] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, Sept 2003, pp. 381–384.

[65] W. C. Wake, *Refactoring Workbook*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[66] M. Usman, R. Britto, J. Börstler, and E. Mendes, "Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method," *Information and Software Technology*, vol. 85, pp. 43 – 59, 2017.

DUPLICATE CODE GROUP (DCG)

[67] C. Kapser and M. Godfrey, "Improved tool support for the investigation of duplication in software," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, 2005, pp. 305–314.

[68] Y. Bian, G. Koru, X. Su, and P. Ma, "SPAPE: A semantic-preserving amorphous procedure extraction method for near-miss clones," *Journal of Systems and Software*, vol. 86, no. 8, pp. 2077–2093, aug 2013.

[69] H. A. Basit and S. Jarzabek, "Detecting Higher-level Similarity Patterns in Programs," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 156–165, 2005.

[70] E. Duala-Ekoko and M. P. Robillard, "Clone Region Descriptors: Representing and Tracking Duplication in Source Code," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 1, pp. 3:1—-3:31, 2010.

[71] Y. Higo and S. Kusumoto, "How Should We Measure Functional Sameness from Program Source Code? An Exploratory Study on Java Methods," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 294–305.

[72] M. Balint, R. Marinescu, and T. Girba, "How Developers Copy," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, 2006, pp. 56–68.

[73] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," *Software: Practice and Experience*, vol. 37, no. 2, pp. 151–175, 2007.

[74] G. Zhang, X. Peng, Z. Xing, and W. Zhao, "Towards contextual and on-demand code clone management by continuous monitoring," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, nov 2013, pp. 497–507.

[75] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*. New York, New York, USA: ACM Press, 2008, p. 321.

[76] J. Ossher, H. Sajnani, and C. Lopes, "File cloning in open source Java projects: The good, the bad, and the ugly," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, sep 2011, pp. 283–292.

[77] C. Roy and J. Cordy, "Scenario-Based Comparison of Clone Detection Techniques," in *2008 16th IEEE International Conference on Program Comprehension*. IEEE, jun 2008, pp. 153–162.

[78] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, mar 2006.

[79] M. Mondal, C. K. Roy, and K. A. Schneider, "Insight into a method co-change pattern to identify highly coupled methods: An empirical study," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, may 2013, pp. 103–112.

[80] W. Wang and M. W. Godfrey, "Recommending Clones for Refactoring Using Design, Context, and History," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, sep 2014, pp. 331–340.

[81] H. Kim, Y. Jung, S. Kim, and K. Yi, "MeCC: memory comparison-based clone detector," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. New York, New York, USA: ACM Press, 2011, p. 301.

[82] D. Lo, L. Jiang, and A. Budi, "Active refinement of clone anomaly reports," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, jun 2012, pp. 397–407.

[83] C. Roy and J. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in *2008 16th IEEE International Conference on Program Comprehension*. IEEE, jun 2008, pp. 172–181.

[84] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon, "Automated scheduling for clone-based refactoring using a competent GA," *Software: Practice and Experience*, vol. 41, no. 5, pp. 521–550, 2011.

[85] C. Kapser and M. Godfrey, ""Cloning Considered Harmful" Considered Harmful," in *2006 13th Working*

Conference on Reverse Engineering. IEEE, 2006, pp. 19–28.

[86] H. Basit and S. Jarzabek, "A Data Mining Approach for Detecting Higher-Level Clones in Software," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 497–514, jul 2009.

[87] H. A. Basit, U. Ali, S. Haque, and S. Jarzabek, "Things structural clones tell that simple clones don't," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, sep 2012, pp. 275–284.

[88] P. Jablonski and D. Hou, "Aiding Software Maintenance with Copy-and-Paste Clone-Awareness," in *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, jun 2010, pp. 170–179.

[89] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *2010 IEEE International Conference on Software Maintenance*. IEEE, sep 2010, pp. 1–9.

[90] H. A. Basit, S. Jarzabek, D. Anh, and M. Low, "Query-based filtering and graphical view generation for clone analysis," in *2008 IEEE International Conference on Software Maintenance*. IEEE, sep 2008, pp. 376–385.

[91] M. Rieger, S. Ducasse, and M. Lanza, "Insights into system-wide code duplication," in *11th Working Conference on Reverse Engineering*. IEEE Comput. Soc, 2004, pp. 100–109.

[92] J. Svajlenko and C. K. Roy, "Evaluating Modern Clone Detection Tools," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, sep 2014, pp. 321–330.

[93] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, may 2007, pp. 96–105.

[94] T. Mende, R. Koschke, and F. Beckwermert, "An evaluation of code similarity identification for the grow-and-prune model," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 2, pp. 143–169, 2009.

[95] J. Krinke, "A Study of Consistent and Inconsistent Changes to Code Clones," in *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, oct 2007, pp. 170–178.

[96] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe, "On the use of clone detection for identifying crosscutting concern code," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 804–818, oct 2005.

[97] Z. Xing, Y. Xue, and S. Jarzabek, "Distilling useful clones by contextual differencing," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, oct 2013, pp. 102–111.

[98] D. Chatterji, J. C. Carver, N. A. Kraft, and J. Harder, "Effects of cloned code on software maintainability: A replicated developer study," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, oct 2013, pp. 112–121.

[99] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei, "Can I clone this piece of code here?" in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*. New York, New York, USA: ACM Press, 2012, p. 170.

[100] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of International Conference on Software Maintenance ICSM-96*. IEEE, 1996, pp. 244–253.

[101] G. M. Selim, K. C. Foo, and Y. Zou, "Enhancing Source-Based Clone Detection Using Intermediate Representation," in *2010 17th Working Conference on Reverse Engineering*. IEEE, oct 2010, pp. 227–236.

[102] d. M. Wit, A. Zaidman, and A. van Deursen, "Managing code clones using dynamic change tracking and resolution," in *2009 IEEE International Conference on Software Maintenance*. IEEE, sep 2009, pp. 169–178.

[103] E. Duala-Ekoko and M. P. Robillard, "Tracking Code Clones in Evolving Software," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, may 2007, pp. 158–167.

[104] Y. Lin, Z. Xing, X. Peng, Y. Liu, J. Sun, W. Zhao, and J. Dong, "Clonepedia: Summarizing Code Clones by Common Syntactic Context for Software Maintenance," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, sep 2014, pp. 341–350.

[105] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 485–495.

[106] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Gapped code clone detection with lightweight source code analysis," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, may 2013, pp. 93–102.

[107] G. M. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, "Studying the Impact of Clones on Software Defects," in *2010 17th Working Conference on Reverse Engineering*. IEEE, oct 2010, pp. 13–21.

[108] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Partial redesign of Java software systems based on clone analysis," in *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*. IEEE Comput. Soc, 1999, pp. 326–336.

[109] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. New York, New York, USA: ACM Press, 2011, p. 311.

[110] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Advanced clone-analysis to support object-oriented system refactoring," in *Proceedings Seventh Working Conference on Reverse Engineering*. IEEE Comput. Soc, 2000, pp. 98–107.

[111] J. Harder and N. Göde, "Cloned code: stable code," *Journal of Software: Evolution and Process*, vol. 25, no. 10, pp. 1063–1088, 2013.

[112] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, sep 2007.

[113] D. Cai and M. Kim, "An Empirical Study of Long-

Lived Code Clones," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, D. Giannakopoulou and F. Orejas, Eds. Springer Berlin Heidelberg, 2011, vol. 6603, pp. 432–446.

[114] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 61–82, 2006.

[115] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto, "Incremental Code Clone Detection: A PDG-based Approach," in *2011 18th Working Conference on Reverse Engineering*. IEEE, oct 2011, pp. 3–12.

[116] W. S. Evans, C. W. Fraser, and F. Ma, "Clone Detection via Structural Abstraction," in *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, oct 2007, pp. 150–159.

[117] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Clone Management for Evolving Software," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1008–1026, sep 2012.

[118] M. Mondal, C. K. Roy, and K. A. Schneider, "A Fine-Grained Analysis on the Evolutionary Coupling of Cloned Code," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, sep 2014, pp. 51–60.

[119] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Clone-Aware Configuration Management," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, nov 2009, pp. 123–134.

[120] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry, "Understanding the evolution of Type-3 clones: An exploratory study," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, may 2013, pp. 139–148.

[121] M. F. Zibran and C. K. Roy, "Conflict-Aware Optimal Scheduling of Code Clone Refactoring: A Constraint Programming Approach," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, jun 2011, pp. 266–269.

[122] C. K. Roy and J. R. Cordy, "An Empirical Study of Function Clones in Open Source Software," in *2008 15th Working Conference on Reverse Engineering*. IEEE, oct 2008, pp. 81–90.

[123] A. Walenstein, N. Jyoti, and A. Lakhotia, "Problems creating task-relevant clone detection reference data," in *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings.* IEEE, 2003, pp. 285–294.

[124] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, may 2007, pp. 106–115.

[125] C. K. Roy and J. R. Cordy, "Near-miss function clones in open source software: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 3, pp. 165–189, 2010.

[126] J. Harder and R. Tiarks, "A controlled experiment on software clones," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, jun 2012, pp. 219–228.

[127] B. S. Baker, "Finding Clones with Dup: Analysis of an Experiment," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 608–621, sep 2007.

[128] R. Geiger, B. Fluri, H. C. Gall, and M. Pinzger, "Relation of Code Clones and Change Couplings," in *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 411–425.

[129] R. Koschke, R. Falke, and P. Frenzel, "Clone Detection Using Abstract Syntax Suffix Trees," in *2006 13th Working Conference on Reverse Engineering*. IEEE, 2006, pp. 253–262.

[130] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Complete and accurate clone detection in graph-based models," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 276–286.

[131] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone Smells in Software Evolution," in *2007 IEEE International Conference on Software Maintenance*. IEEE, oct 2007, pp. 24–33.

[132] L. Jiang, Z. Su, and E. Chiu, "Context-based Detection of Clone-related Bugs," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 55–64.

[133] W. Qian, X. Peng, Z. Xing, S. Jarzabek, and W. Zhao, "Mining Logical Clones in Software: Revealing High-Level Business and Programming Rules," in *2013 IEEE International Conference on Software Maintenance*. IEEE, sep 2013, pp. 40–49.

[134] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, May 2010, pp. 72–81.

[135] F. Al-Omari, I. Keivanloo, C. K. Roy, and J. Rilling, "Detecting Clones Across Microsoft .NET Programming Languages," in *2012 19th Working Conference on Reverse Engineering*. IEEE, oct 2012, pp. 405–414.

[136] J. Guo and Y. Zou, "Detecting Clones in Business Applications," in *2008 15th Working Conference on Reverse Engineering*. IEEE, oct 2008, pp. 91–100.

[137] J. R. Cordy, "Exploring Large-Scale System Similarity Using Incremental Clone Detection and Live Scatterplots," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, jun 2011, pp. 151–160.

[138] E. Lan, "Predicting Consistency-Maintenance Requirement of Code Clonesat Copy-and-Paste Time," *IEEE Transactions on Software Engineering*, vol. 40, no. 8, pp. 773–794, aug 2014.

[139] G. Zhang, X. Peng, Z. Xing, and W. Zhao, "Cloning practices: Why developers clone and what can be changed," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, sep 2012, pp. 285–294.

[140] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for*

Business Change' (Cat. No.99CB36360). IEEE, 1999, pp. 109–118.

[141] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, "On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems," in *2011 18th Working Conference on Reverse Engineering*. IEEE, oct 2011, pp. 13–22.

[142] S. Xie, F. Khomh, and Y. Zou, "An empirical study of the fault-proneness of clone mutation and clone migration," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 149–158.

[143] D. M. German, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, may 2009, pp. 81–90.

[144] H. Li and S. Thompson, "Incremental Clone Detection and Elimination for Erlang Programs," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, D. Giannakopoulou and F. Orejas, Eds. Springer Berlin Heidelberg, 2011, vol. 6603, pp. 356–370.

[145] C.-H. Wang and F.-J. Wang, "Detecting artifact anomalies in business process specifications with a formal model," *Journal of Systems and Software*, vol. 82, no. 10, pp. 1600–1619, oct 2009.

[146] S. Bazrafshan and R. Koschke, "An Empirical Study of Clone Removals," in *2013 IEEE International Conference on Software Maintenance*. IEEE, sep 2013, pp. 50–59.

[147] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings Eighth Working Conference on Reverse Engineering*. IEEE Comput. Soc, 2001, pp. 301–309.

[148] B. Hauptmann, M. Junker, S. Eder, E. Juergens, and R. Vaas, "Can clone detection support test comprehension?" in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, jun 2012, pp. 209–218.

[149] M. Bruntink, A. van Deursen, T. Tourwe, and R. van Engelen, "An evaluation of clone detection techniques for identifying crosscutting concerns," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 200–209.

[150] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *2008 IEEE International Conference on Software Maintenance*. IEEE, sep 2008, pp. 227–236.

[151] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen, "Accurate and Efficient Structural Characteristic Feature Extraction for Clone Detection," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, M. Chechik and M. Wirsing, Eds. Springer Berlin Heidelberg, 2009, vol. 5503, pp. 440–455.

[152] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE Comput. Soc, 1998, pp. 368–377.

[153] R. Fanta and V. Rajlich, "Removing clones from the code," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 4, pp. 223–243, 1999.

[154] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 1, pp. 37–58, 2006.

[155] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, jul 2002.

[156] L. Barbour, F. Khomh, and Y. Zou, "Late propagation in software clones," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, sep 2011, pp. 273–282.

[157] J. Harder, "How Multiple Developers Affect the Evolution of Code Clones," in *2013 IEEE International Conference on Software Maintenance*. IEEE, sep 2013, pp. 30–39.

[158] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe, "Where does this code come from and where does it go? — Integrated code history tracker for open source systems," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, jun 2012, pp. 331–341.

[159] J. Li and M. D. Ernst, "CBCD: Cloned buggy code detector," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, jun 2012, pp. 310–320.

[160] R. K. Saha, C. K. Roy, and K. A. Schneider, "An automatic framework for extracting and classifying near-miss clone genealogies," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, sep 2011, pp. 293–302.

[161] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An Empirical Study on Inconsistent Changes to Code Clones at Release Level," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 85–94.

[162] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An Exploratory Study of Cloning in Industrial Software Product Lines," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2013, pp. 25–34.

[163] R. Koschke, "Large-Scale Inter-System Clone Detection Using Suffix Trees," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2012, pp. 309–318.

[164] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner, "Challenges of the Dynamic Detection of Functionally Similar Code Fragments," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2012, pp. 299–308.

[165] Y. Higo and S. Kusumoto, "Code Clone Detection on Specialized PDGs with Heuristics," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2011, pp. 75–84.

[166] N. Gode and J. Harder, "Clone Stability," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2011, pp. 65–74.

[167] E. Juergens, F. Deissenboeck, and B. Hummel, "Code Similarities Beyond Copy & Paste," in *2010 14th European Conference on Software Maintenance and Reengi-*

neering. IEEE, mar 2010, pp. 78–87.

[168] N. Göde and R. Koschke, "Incremental Clone Detection," in *2009 13th European Conference on Software Maintenance and Reengineering*. IEEE, 2009, pp. 219–228.

[169] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection," in *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, apr 2008, pp. 163–172.

[170] L. Aversano, L. Cerulo, and M. Di Penta, "How Clones are Maintained: An Empirical Study," in *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE, 2007, pp. 81–90.

[171] H. Sajnani, V. Saini, and C. V. Lopes, "A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2014, pp. 21–30.

[172] M. Mondal, C. K. Roy, and K. A. Schneider, "Automatic Identification of Important Clones for Refactoring and Tracking," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2014, pp. 11–20.

[173] S. Bazrafshan and R. Koschke, "Effect of Clone Information on the Performance of Developers Fixing Cloned Bugs," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2014, pp. 1–10.

[174] M. S. Rahman and C. K. Roy, "A Change-Type Based Empirical Study on the Stability of Cloned Code," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2014, pp. 31–40.

[175] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue, "Identifying Source Code Reuse across Repositories Using LCS-Based Source Code Similarity," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2014, pp. 305–314.

[176] H. Murakami, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Folding Repeated Instructions for Improving Token-Based Code Clone Detection," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2012, pp. 64–73.

[177] S. Bazrafshan, "Evolution of Near-Miss Clones," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2012, pp. 74–83.

[178] W. Wang and M. W. Godfrey, "A Study of Cloning in the Linux SCSI Drivers," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2011, pp. 95–104.

[179] S. Schulze, E. Jurgens, and J. Feigenspan, "Analyzing the Effect of Preprocessor Annotations on Code Clones," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2011, pp. 115–124.

[180] M. F. Zibran and C. K. Roy, "A Constraint Programming Approach to Conflict-Aware Optimal Scheduling of Prioritized Code Clone Refactoring," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2011, pp. 105–114.

[181] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating Code Clone Genealogies at Release Level: An Empirical Study," in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2010, pp. 87–96.

[182] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes, "Language-Independent Clone Detection Applied to Plagiarism Detection," in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2010, pp. 77–86.

[183] R. Tiarks, R. Koschke, and R. Falke, "An Assessment of Type-3 Clones as Detected by State-of-the-Art Tools," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009, pp. 67–76.

[184] N. Göde, "Evolution of Type-1 Clones," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2009, pp. 77–86.

[185] J. Krinke, "Is Cloned Code More Stable than Non-cloned Code?" in *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2008, pp. 57–66.

[186] Z. M. Jiang and A. E. Hassan, "A Framework for Studying Clones In Large Software Systems," in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*. IEEE, sep 2007, pp. 203–212.

[187] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, and D. Poshyvanyk, "Revisiting Android Reuse Studies in the Context of Code Obfuscation and Library Usages," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 242–251.

[188] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, and N. A. Kraft, "Measuring the Efficacy of Code Clone Information in a Bug Localization Task: An Empirical Study," in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, 2011, pp. 20–29.

[189] M. Shomrat and Y. Feldman, "Detecting Refactored Clones," in *ECOOP 2013 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, G. Castagna, Ed. Springer Berlin Heidelberg, 2013, vol. 7920, pp. 502–526.

[190] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su, "Scalable and Systematic Detection of Buggy Inconsistencies in Source Code," *SIGPLAN Not.*, vol. 45, no. 10, pp. 175–190, 2010.

[191] E. R. Murphy-Hill, P. J. Quitslund, and A. P. Black, "Removing Duplication from Java.Io: A Case Study Using Traits," in *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 282–291.

[192] F. Rahman, C. Bird, and P. Devanbu, "Clones: what is that smell?" *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 503–530, 2012.

[193] C. J. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, jul 2008.

[194] C. Parnin, C. Bird, and E. Murphy-Hill, "Adoption and use of Java generics," *Empirical Software Engineering*, vol. 18, no. 6, pp. 1047–1089, 2013.

[195] R. Tairas and J. Gray, "An information retrieval process to aid in the analysis of code clones," *Empirical Software Engineering*, vol. 14, no. 1, pp. 33–56, 2009.

[196] R. Falke, P. Frenzel, and R. Koschke, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, jul 2008.

[197] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, no. 1, pp. 1–34, 2010.

[198] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting Differences Across Multiple Instances of Code Clones," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 164–174.

[199] K. Chen, P. Liu, and Y. Zhang, "Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 175–186.

[200] M. Mondal, C. K. Roy, and K. A. Schneider, "An Empirical Study on Clone Stability," *SIGAPP Appl. Comput. Rev.*, vol. 12, no. 3, pp. 20–36, 2012.

[201] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie, "XIAO: Tuning Code Clones at Hands of Engineers in Practice," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 369–378.

[202] C. Arwin and S. M. M. Tahaghoghi, "Plagiarism Detection Across Programming Languages," in *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ser. ACSC '06. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 277–286.

[203] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "On detection of gapped code clones using gap locations," in *Software Engineering Conference, 2002. Ninth Asia-Pacific*, 2002, pp. 327–336.

[204] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the relation between changeability decay and the characteristics of clones and methods," in *Automated Software Engineering - Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, 2008, pp. 100–109.

[205] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Journal Automated Software Engineering*, vol. 3, no. 1-2, pp. 77–108, 1996.

[206] J. R. Cordy, T. R. Dean, and N. Synytskyy, "Practical Language-independent Detection of Near-miss Clones," in *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '04. IBM Press, 2004, pp. 1–12.

[207] S. Xie, F. Khomh, Y. Zou, and I. Keivanloo, "An empirical study on the fault-proneness of clone migration in clone genealogies," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, 2014, pp. 94–103.

[208] M. Mandal, C. K. Roy, and K. A. Schneider, "Automatic ranking of clones for refactoring through mining association rules," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, 2014, pp. 114–123.

[209] G. P. Krishnan and N. Tsantalis, "Unification and refactoring of clones," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, 2014, pp. 104–113.

[210] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future (Keynote paper)," in *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, 2014, pp. 18–33.

[211] B. S. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, pp. 49–49, 1993.

[212] E. Merlo, "Detection of Plagiarism in University Projects Using Metrics-based Spectral Similarity," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no. 06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum f{ü}r Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[213] S. Giesecke, "Generic modelling of code clones," in *Duplication, Redundancy, and Similarity in Software*, ser. Dagstuhl Seminar Proceedings, R. Koschke, E. Merlo, and A. Walenstein, Eds., no. 06301. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum f{ü}r Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[214] R. Koschke, "Survey of research on software clones," *Dublication, Redundancy, and Similarity in Software - Dagstuhl Seminar #06301*, p. 24, 2007.

[215] T. R. Dean, J. Chen, and M. H. Alalfi, "Clone Detection in Matlab Stateflow Models," *ECEASST*, vol. 63, 2014.

[216] H. Störrle, "Towards Clone Detection in UML Domain Models," in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ser. ECSA '10. New York, NY, USA: ACM, 2010, pp. 285–293.

[217] C. Kapser and M. W. Godfrey, "Toward a taxonomy of clones in source code: A case study," in *Proceedings of the Conference on Evolution of Large Scale Industrial Software Architectures (ELISA'03)*, 2003, pp. 67–78.

[218] R. Koschke, "Frontiers of software clone management," in *Frontiers of Software Maintenance, 2008. FoSM 2008.*, 2008, pp. 119–128.

[219] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An Empirical Study of Code Clone Genealogies," *SIG-*

SOFT Softw. Eng. Notes, vol. 30, no. 5, pp. 187–196, 2005.

[220] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim, "Instant Code Clone Search," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 167–176.

[221] M. F. Zibran, R. K. Saha, M. Asaduzzaman, and C. K. Roy, "Analyzing and Forecasting Near-Miss Clones in Evolving Software: An Empirical Study," in *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, 2011, pp. 295–304.

[222] H. Basit, D. Rajapakse, and S. Jarzabek, "Beyond templates: a study of clones in the STL and some general implications," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* IEEe, 2005, pp. 451–459.

[223] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert, "Clone detection in automotive model-based development," in *Proceedings of the 13th international conference on Software engineering - ICSE '08*. New York, New York, USA: ACM Press, 2008, p. 603.

[224] D. C. Rajapakse and S. Jarzabek, "Using Server Pages to Unify Clones in Web Applications: A Trade-Off Analysis," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, may 2007, pp. 116–126.

[225] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting Differences Across Multiple Instances of Code Clones," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 164–174.

[226] M. H. Alalfi, J. R. Cordy, T. R. Dean, M. Stephan, and A. Stevenson, "Models are code too: Near-miss clone detection for Simulink models," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, sep 2012, pp. 295–304.

[227] C. K. Roy and J. R. Cordy, "A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools," in *Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on*, 2009, pp. 157–166.

[228] C. K. Roy, "Conflict-aware optimal scheduling of prioritised code clone refactoring," *IET Software*, vol. 7, no. 3, pp. 167–186(19), 2013.

[229] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," in *Empirical Software Engineering, 2004. ISESE '04. Proceedings. 2004 International Symposium on*, 2004, pp. 83–92.

[230] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey, "Cloning by accident: an empirical study of source code cloning across software systems," in *Empirical Software Engineering, 2005. 2005 International Symposium on*, nov 2005, pp. 10 pp.–.

[231] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting Code Clones in Binary Executables," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 117–128.

[232] G. Antoniol, U. Villano, E. Merlo, and M. D. Penta, "Analyzing cloning evolution in the Linux kernel," *Information and Software Technology*, vol. 44, no. 13, pp. 755–765, 2002.

[233] R. Tairas and J. Gray, "Increasing Clone Maintenance Support by Unifying Clone Detection and Refactoring Activities," *Inf. Softw. Technol.*, vol. 54, no. 12, pp. 1297–1307, 2012.

[234] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Method and Implementation for Investigating Code Clones in a Software System," *Inf. Softw. Technol.*, vol. 49, no. 9-10, pp. 985–998, 2007.

[235] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[236] R. Tiarks, R. Koschke, and R. Falke, "An extended assessment of type-3 clones as detected by state-of-the-art tools," *Software Quality Journal*, vol. 19, no. 2, pp. 295–331, 2011.

[237] J. Svajlenko, I. Keivanloo, and C. K. Roy, "Big data clone detection using classical detectors: an exploratory study," *Journal of Software: Evolution and Process*, vol. 27, no. 6, pp. 430–464, 2015.

[238] J.-w. Park, M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim, "Surfacing code in the dark: an instant clone search approach," *Knowledge and Information Systems*, vol. 41, no. 3, pp. 727–759, 2014.

[239] Y. Higo, S. Kusumoto, and K. Inoue, "A Metric-based Approach to Identifying Refactoring Opportunities for Merging Code Clones in a Java Software System," *J. Softw. Maint. Evol.*, vol. 20, no. 6, pp. 435–461, 2008.

[240] L. Barbour, F. Khomh, and Y. Zou, "An empirical study of faults in late propagation clone genealogies," *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1139–1165, 2013.

[241] S. Jarzabek and S. Li, "Unifying clones with a generative programming technique: a case study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 4, pp. 267–292, 2006.

[242] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[243] Y. Higo, S. Kusumoto, and K. Inoue, "Identifying refactoring opportunities for removing code clones with a metrics-based approach," *Java in Academia and Research*, pp. 57–82, 2011.

[244] R. Koschke, "Identifying and Removing Software Clones," in *Software Evolution*. Springer Berlin Heidelberg, 2008, pp. 15–36.

[245] F. Calefato, F. Lanubile, and T. Mallardo, "Function Clone Detection in Web Applications: A Semiautomated Approach," *J. Web Eng.*, vol. 3, no. 1, pp. 3–21, 2004.

[246] I. Keivanloo, C. K. Roy, and J. Rilling, "SeByte: Scalable clone and similarity search for bytecode," *Science of Computer Programming*, vol. 95, Part 4, pp. 426–444, 2014.

[247] N. Davey, P. C. Barson, S. D. H. Field, R. J. Frank, and D. S. W. Tansley, "The Development of a Software Clone Detector," 1995.

[248] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *Static Analysis*, ser. Lecture Notes in Computer Science, P. Cousot, Ed. Springer Berlin Heidelberg, 2001, vol. 2126, pp. 40–56.

[249] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An Empirical Study on Inconsistent Changes to Code Clones at Release Level," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 85–94.

[250] H. Li and S. Thompson, "Similar Code Detection and Elimination for Erlang Programs," in *Proceedings of the 12th International Conference on Practical Aspects of Declarative Languages*, ser. PADL'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 104–118.

[251] B. Al-Batran, B. Schätz, and B. Hummel, "Semantic Clone Detection for Model-based Development of Embedded Systems," in *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 258–272.

[252] H. A. Basit, D. C. Rajapakse, and S. Jarzabek, "An Empirical Study on Limits of Clone Unification Using Generics," in *In Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05*, 2005, pp. 109–114.

[253] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Refactoring Support Based on Code Clone Analysis," in *Product Focused Software Process Improvement*, ser. Lecture Notes in Computer Science, F. Bomarius and H. Iida, Eds. Springer Berlin Heidelberg, 2004, vol. 3009, pp. 220–233.

[254] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, p. 20.

[255] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "On Software Maintenance Process Improvement Based on Code Clone Analysis," in *Proceedings of the 4th International Conference on Product Focused Software Process Improvement*, ser. PROFES '02. London, UK, UK: Springer-Verlag, 2002, pp. 185–197.

[256] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities," in *Software Metrics Symposium, 1999. Proceedings. Sixth International*, 1999, pp. 292–303.

[257] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Matsumoto, and H. Kudo, "Software analysis by code clones in open source software," *Journal of Computer Information Systems*, vol. 45, no. 3, pp. 1–11, 2005.

[258] A. De Lucia, R. Francese, G. Scanniello, and G. Tortora, "Reengineering web applications based on cloned pattern analysis," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 2004, pp. 132–141.

[259] C. Kapser and M. W. Godfrey, "Aiding Comprehension of Cloning Through Categorization," in *Proceedings of the Principles of Software Evolution, 7th International Workshop*, ser. IWPSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 85–94.

[260] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is Duplicate Code More Frequently Modified Than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ser. IWPSE-EVOL '10. New York, NY, USA: ACM, 2010, pp. 73–82.

[261] C. Brown and S. Thompson, "Clone Detection and Elimination for Haskell," in *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '10. New York, NY, USA: ACM, 2010, pp. 111–120.

[262] E. Juergens and F. Deissenboeck, "How much is a clone," in *Proceedings of the 4th International Workshop on Software Quality and Maintainability*, 2010.

[263] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: maintenance support environment based on code clone analysis," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, 2002, pp. 67–76.

[264] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "On refactoring support based on code clone dependency relation," in *Software Metrics, 2005. 11th IEEE International Symposium*, 2005, pp. 10 pp.–16.

[265] B. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of 2nd Working Conference on Reverse Engineering*. IEEE Comput. Soc. Press, 1995, pp. 86–95.

[266] A. Hemel and R. Koschke, "Reverse Engineering Variability in Source Code Using Clone Detection: A Case Study for Linux Variants of Consumer Electronic Devices," in *2012 19th Working Conference on Reverse Engineering*. IEEE, oct 2012, pp. 357–366.

[267] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient Similarity Joins for Near Duplicate Detection," in *Proceedings of the 17th International Conference on World Wide Web*, ser. WWW '08. New York, NY, USA: ACM, 2008, pp. 131–140.

[268] D. Chatterji, J. C. Carver, and N. A. Kraft, "Code clones and developer behavior: results of two surveys of the clone research community," *Empirical Software Engineering*, vol. 21, no. 4, pp. 1476–1508, 2016.

[269] I. Keivanloo, F. Zhang, and Y. Zou, "Threshold-free code clone detection for a large-scale heterogeneous Java repository," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, mar 2015, pp. 201–210.

[270] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 87–98.

[271] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Classification model for code clones based on machine learning," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1095–1125, 2015.

[272] W. T. Cheung, S. Ryu, and S. Kim, "Development nature matters: An empirical study of code clones in javascript applications," *Empirical Software Engineering*, vol. 21, no. 2, pp. 517–564, 2016.

[273] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations: A rigorous approach to clone evaluation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 455–465.

[274] Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao, "Clone-based and interactive recommendation for modifying pasted code," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 520–531.

[275] F.-H. Su, J. Bell, G. Kaiser, and S. Sethumadhavan, "Identifying functionally similar code in complex codebases," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[276] X. Cheng, H. Zhong, Y. Chen, Z. Hu, and J. Zhao, "Rule-directed code clone synchronization," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[277] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1157–1168.

[278] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 131–140.

[279] M. Mondal, C. K. Roy, and K. A. Schneider, "A comparative study on the bug-proneness of different types of code clones," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 91–100.

[280] V. Saini, H. Sajnani, and C. Lopes, "Comparing quality metrics for cloned and non cloned java methods: A large scale empirical study," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 256–266.

[281] H. Sajnani, V. Saini, and C. Lopes, "A parallel and efficient approach to large scale clone detection," *Journal of Software: Evolution and Process*, vol. 27, no. 6, pp. 402–429, 2015, jSME-13-0129.R2.

[282] R. Ettinger, S. Tyszberowicz, and S. Menaia, "Efficient method extraction for automatic elimination of type-3 clones," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 327–337.

[283] M. S. Uddin, V. Gaur, C. Gutwin, and C. K. Roy, "On the comprehension of code clone visualizations: A controlled study using eye tracking," in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sept 2015, pp. 161–170.

[284] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "Similarity of source code in the presence of pervasive modifications," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Oct 2016, pp. 117–126.

[285] N. Tsantalis, D. Mazinanian, and G. P. Krishnan, "Assessing the refactorability of software clones," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1055–1090, Nov 2015.

[286] D. Mazinanian, N. Tsantalis, and A. Mesbah, "Discovering refactoring opportunities in cascading style sheets," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 496–506.

[287] M. Mondal, C. K. Roy, and K. A. Schneider, "Prediction and ranking of co-change candidates for clones," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 32–41.

[288] T. M. Ahmed, W. Shang, and A. E. Hassan, "An empirical study of the copy and paste behavior during development," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 99–110.

[289] N. Göde and R. Koschke, "Studying clone evolution using incremental clone detection," *Journal of Software: Evolution and Process*, vol. 25, no. 2, pp. 165–192, 2013.

[290] A. Charpentier, J.-R. Falleri, D. Lo, and L. Réveillère, "An empirical assessment of bellon's clone benchmark," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '15. New York, NY, USA: ACM, 2015, pp. 20:1–20:10.

[291] M. Mondal, C. K. Roy, and K. A. Schneider, "An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study," *Science of Computer Programming*, vol. 95, Part 4, pp. 445 – 468, 2014, special Issue on Software Clones (IWSC'12).

[292] J. F. Islam, M. Mondal, and C. K. Roy, "Bug replication in code clones: An empirical study," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 68–78.

[293] S. Wagner, A. Abdulkhaleq, K. Kaya, and A. Paar, "On the relationship of inconsistent software clones and faults: An empirical study," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 79–89.

OTHER BAD SMELLS GROUP (OBSG)

[294] I. Macia Bertran, A. Garcia, and A. von Staa, "An Exploratory Study of Code Smells in Evolving Aspect-oriented Systems," in *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, ser. AOSD '11. New York, NY, USA: ACM, 2011, pp. 203–214.

[295] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *2010 10th International Conference on Quality Software*, July 2010, pp. 23–31.

[296] H. Liu, Z. Ma, W. Shao, and Z. Niu, "Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 220–235, jan 2012.

[297] N. Sae-Lim, S. Hayashi, and M. Saeki, "Context-based code smells prioritization for prefactoring," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[298] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 315–326.

[299] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka, "Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains," in *2013 IEEE International Conference on Software Maintenance*. IEEE, sep 2013, pp. 260–269.

[300] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design Defects Detection and Correction by Example," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, jun 2011, pp. 81–90.

[301] S. Fu and B. Shen, "Code bad smell detection through evolutionary data mining," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Oct 2015, pp. 1–9.

[302] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. Ben Chikha, "Competitive coevolutionary code-smells detection," in *Proceedings of the 5th International Symposium on Search Based Software Engineering - Volume 8084*, ser. SSBSE 2013. New York, NY, USA: Springer-Verlag New York, Inc., 2013, pp. 50–65.

[303] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Journal Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[304] F. Hermans and E. Aivaloglou, "Do code smells hamper novice programming? a controlled experiment on scratch programs," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[305] P. F. Mihancea and R. Marinescu, "Towards the optimization of automatic detection of design flaws in object-oriented software systems," in *Ninth European Conference on Software Maintenance and Reengineering*, March 2005, pp. 92–101.

[306] M. Salehie, S. Li, and L. Tahvildari, "A metric-based heuristic framework to detect object-oriented design flaws," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, 2006, pp. 159–168.

[307] T. Tourwe and T. Mens, "Identifying refactoring opportunities using logic meta programming," in *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, March 2003, pp. 91–100.

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

[309] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[310] H. Liu, Q. Liu, Z. Niu, and Y. Liu, "Dynamic and automatic feedback-based threshold adaptation for code smell detection," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 544–558, June 2016.

[311] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, "An Exploratory Study of the Impact of Code Smells on Software Change-proneness," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 75–84.

[312] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, no. 3, pp. 395–431, may 2006.

[313] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, sep 2012, pp. 306–315.

[314] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, and Y.-G. Guéhéneuc, "Domain Matters: Bringing Further Evidence of the Relationships Among Anti-patterns, Application Domains, and Quality-related Metrics in Java Mobile Apps," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 232–243.

[315] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are Automatically-detected Code Anomalies Relevant to Architectural Modularity?: An Exploratory Analysis of Evolving Systems," in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, ser. AOSD '12. New York, NY, USA: ACM, 2012, pp. 167–178.

[316] I. Macia, A. Garcia, A. von Staa, J. Garcia, and N. Medvidovic, "On the Impact of Aspect-Oriented Code Smells on Architecture Modularity: An Exploratory Study," in *Software Components, Architectures and Reuse (SBCARS), 2011 Fifth Brazilian Symposium on*, 2011, pp. 41–50.

[317] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan, "Predicting Bugs Using Antipatterns," in *2013 IEEE International Conference on Software Maintenance*. IEEE, sep 2013, pp. 270–279.

[318] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa, "On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms," in *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2012, pp. 277–286.

[319] F. Khomh, M. D. Penta, Y.-G. Gu'eh'eneuc, and G. Antoniol, "An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness," *Empirical Softw. Engg.*, vol. 17, no. 3, pp. 243–275, 2012.

[320] W. Oizumi, A. Garcia, M. Ferreira, A. von Staa, and T. E. Colanzi, "When Code-Anomaly Agglomerations Represent Architectural Problems? An Exploratory Study," in *Software Engineering (SBES), 2014 Brazilian Symposium on*, 2014, pp. 91–100.

[321] I. Ahmed, U. A. Mannan, R. Gopinath, and C. Jensen, "An empirical study of design degradation: How soft-

ware projects get worse over time," in *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Oct 2015, pp. 1–10.

[322] A. Yamashita and S. Counsell, "Code smells as system-level indicators of maintainability: An empirical study," *Journal of Systems and Software*, vol. 86, no. 10, pp. 2639–2653, oct 2013.

[323] D. I. Sjoberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dyba, "Quantifying the Effect of Code Smells on Maintenance Effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, aug 2013.

[324] A. Yamashita, "Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative and Qualitative Data," *Empirical Softw. Engg.*, vol. 19, no. 4, pp. 1111–1143, 2014.

[325] D. Romano, P. Raila, M. Pinzger, and F. Khomh, "Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes," in *2012 19th Working Conference on Reverse Engineering*. IEEE, oct 2012, pp. 437–446.

[326] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, sep 2014, pp. 101–110.

[327] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, may 2013, pp. 682–691.

[328] Y. Aiko and M. Leon, "To What Extent Can Maintenance Problems Be Predicted by Code Smell Detection? - An Empirical Study," *Inf. Softw. Technol.*, vol. 55, no. 12, pp. 2223–2242, 2013.

[329] A. Yamashita, M. Zanoni, F. A. Fontana, and B. Walter, "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)*, Sept 2015, pp. 121–130.

[330] Z. Soh, A. Yamashita, F. Khomh, and Y. G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?" in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 393–402.

[331] F. Jaafar, Y.-G. Gueheneuc, S. Hamel, and F. Khomh, "Mining the relationship between anti-patterns dependencies and fault-proneness," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, oct 2013, pp. 351–360.

[332] F. Jaafar, Y.-g. Guéhéneuc, S. Hamel, and K. Foutse, "Analysing Anti-patterns Static Relationships with Design Patterns," Electronic Communications of the EASST (ECEASST), 2013.

[333] F. Palomba, A. D. Lucia, G. Bavota, and R. Oliveto, "Anti-pattern detection: Methods, challenges, and open issues." *Advances in Computers*, vol. 95, pp. 201–238, 2015.

[334] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh, and M. Zulkernine, "Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults," *Empirical Software Engineering*, vol. 21, no. 3, pp. 896–931, 2016.

[335] N. Moha, Y.-G. Guéhéneuc, A.-F. Meur, L. Duchien, and A. Tiberghien, "From a domain analysis to the specification and detection of code and design smells," *Formal Aspects of Computing*, vol. 22, no. 3-4, pp. 345–361, 2010.

[336] C. Parnin, C. Görg, and O. Nnadi, "A Catalogue of Lightweight Visualizations to Support Code Smell Inspection," in *Proceedings of the 4th ACM Symposium on Software Visualization*, ser. SoftVis '08. New York, NY, USA: ACM, 2008, pp. 77–86.

[337] N. Zazworka, A. Vetro', C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull, "Comparing four approaches for technical debt identification," *Software Quality Journal*, vol. 22, no. 3, pp. 403–426, 2014.

[338] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Journal Automated Software Engineering*, vol. 23, no. 3, pp. 501–532, 2016.

[339] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 350–359.

[340] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, sep 2014.

[341] R. Marinescu, "Assessing technical debt by identifying design flaws in software systems," *IBM Journal of Research and Development*, vol. 56, no. 5, pp. 9:1–9:13, Sept 2012.

[342] E. Murphy-Hill and A. P. Black, "An interactive ambient visualization for code smells," in *Proceedings of the 5th International Symposium on Software Visualization*, ser. SOFTVIS '10. New York, NY, USA: ACM, 2010, pp. 5–14.

[343] H. Liu, X. Guo, and W. Shao, "Monitor-Based Instant Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1112–1126, aug 2013.

[344] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 171–180.

[345] I. Macia, A. Garcia, C. Chavez, and A. von Staa, "Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2013, pp. 177–186.

[346] W. Oizumi, A. Garcia, L. da Silva Sousa, B. Cafeo, and Y. Zhao, "Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 440–451.

[347] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-smell detection as a bilevel problem," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 6:1–6:44,

Oct. 2014.

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

[349] W. N. Oizumi, A. F. Garcia, T. E. Colanzi, M. Ferreira, and A. V. Staa, "On the relationship of code-anomaly agglomerations and architectural problems," *Journal of Software Engineering Research and Development*, vol. 3, no. 1, p. 11, 2015.

[350] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment." *Journal of Object Technology*, vol. 11, no. 2, pp. 1–5, 2012.

[351] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 4–15.

[352] F. Palomba, M. Zanoni, F. A. Fontana, A. D. Lucia, and R. Oliveto, "Smells like teen spirit: Improving bug prediction performance using the intensity of code smells," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 244–255.

[353] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, "Towards a prioritization of code debt: A code smell intensity index," in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, Oct 2015, pp. 16–24.

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

[355] A. M. Fard and A. Mesbah, "JSNOSE: Detecting JavaScript Code Smells," in *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, sep 2013, pp. 116–125.

[356] R. Fourati, N. Bouassida, and H. Abdallah, "A Metric-Based Approach for Anti-pattern Detection in UML Designs," in *Computer and Information Science 2011*, ser. Studies in Computational Intelligence, R. Lee, Ed. Springer Berlin Heidelberg, 2011, vol. 364, pp. 17–33.

[357] T. Hall, M. Zhang, D. Bowes, and Y. Sun, "Some Code Smells Have a Significant but Small Effect on Faults," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 33:1—-33:39, 2014.

[358] S. Peldszus, G. Kulcsár, M. Lochau, and S. Schulze, "Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sept 2016, pp. 578–589.

[359] F. Palomba, A. Panichella, A. D. Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[360] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 403–414.

[361] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, May 2015.

[362] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *Proceedings of the Sixth International Workshop on Emerging Trends in Software Metrics*, ser. WETSoM '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 44–53.

[363] R. Morales, A. Sabane, P. Musavi, F. Khomh, F. Chicano, and G. Antoniol, "Finding the best compromise between design quality and testing effort during refactoring," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 24–35.

[364] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur, "SMURF: A SVM-based Incremental Anti-pattern Detection Approach," in *2012 19th Working Conference on Reverse Engineering*. IEEE, oct 2012, pp. 466–475.

[365] N. Moha, Y.-G. Gu'eh'eneuc, A.-F. Le Meur, and L. Duchien, "A Domain Analysis to Specify Design Defects and Generate Detection Algorithms," in *Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 276–291.

[366] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, jan 2010.

[367] R. Marinescu and C. Marinescu, "Are the Clients of Flawed Classes (Also) Defect Prone?" in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2011, pp. 65–74.

[368] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the software quality of android applications along their evolution (t)," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 236–247.

[369] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of code smells in object-oriented systems," *Innovations in Systems and Software Engineering*, vol. 10, no. 1, pp. 3–18, 2014.

[370] B. Karasneh, M. R. V. Chaudron, F. Khomh, and Y. G. Gueheneuc, "Studying the relation between anti-patterns in design models and in source code," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 36–45.

[371] M. Mäntylä, J. Vanhanen, and C. Lassenius, "Bad smells -humans as code critics," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* IEEE, 2004, pp. 399–408.

[372] A. Yamashita and L. Moonen, "Do developers care about code smells? An exploratory survey," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, oct 2013, pp. 242–251.

[373] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, apr 2011.

[374] N. Tsantalis and A. Chatzigeorgiou, "Ranking Refactoring Suggestions Based on Historical Volatility," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2011, pp. 25–34.

[375] B. Pietrzak and B. Walter, "Leveraging Code Smell Detection with Inter-smell Relations," in *Proceedings of the 7th International Conference on Extreme Programming and Agile Processes in Software Engineering*, ser. XP'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 75–84.

[376] A. Chatzigeorgiou and A. Manakos, "Investigating the Evolution of Bad Smells in Object-Oriented Code," in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, 2010, pp. 106–115.

[377] G. d. F. Carneiro, M. Silva, L. Mara, E. Figueiredo, C. Sant'Anna, A. Garcia, and M. Mendonca, "Identifying code smells with multiple concern views," in *2010 Brazilian Symposium on Software Engineering*, Sept 2010, pp. 128–137.

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

[379] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE Comput. Soc, 2002, pp. 97–106.

[380] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The Evolution and Impact of Code Smells: A Case Study of Two Open Source Systems," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 390–400.

[381] D. Steidl and S. Eder, "Prioritizing Maintainability Defects Based on Refactoring Recommendations," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 168–176.

[382] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension," in *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2011, pp. 181–190.

[383] V. Ferme, A. Marino, and F. A. Fontana, "Is it a Real Code Smell to be Removed or not?" in *International Workshop on Refactoring & Testing (RefTest), co-located event with XP 2013 Conference*, 2013.

[384] M. J. Munro, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code," in *Software Metrics, 2005. 11th IEEE International Symposium*, 2005, p. 15.

[385] D. Rapu, S. Ducasse, T. Girba, and R. Marinescu, "Using history information to improve design flaws detection," in *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, March 2004, pp. 223–232.

[386] J. Kreimer, "Adaptive detection of design flaws," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117 – 136, 2005.

[387] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Gueheneuc, "Tracking Design Smells: Lessons from a Study of God Classes," in *2009 16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 145–154.

[388] H. C. Jiau and J. C. Chen, "OBEY: Optimal Batched Refactoring Plan Execution for Class Responsibility Redistribution," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1245–1263, sep 2013.

[389] A. De Lucia, R. Oliveto, and L. Vorraro, "Using structural and semantic metrics to improve class cohesion," in *2008 IEEE International Conference on Software Maintenance*. IEEE, sep 2008, pp. 27–36.

[390] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of Extract Class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, oct 2012.

[391] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending Move Method Refactorings via Relational Topic Models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, jul 2014.

[392] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending Move Method refactorings using dependency sets," in *2013 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, oct 2013, pp. 232–241.

[393] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, may 2009.

[394] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, sep 2012, pp. 56–65.

[395] S. L. Abebe, V. Arnaoudova, P. Tonella, G. Antoniol, and Y.-G. Gueheneuc, "Can Lexicon Bad Smells Improve Fault Prediction?" in *2012 19th Working Conference on Reverse Engineering*. IEEE, oct 2012, pp. 235–244.

[396] G. Bavota, A. De Lucia, and R. Oliveto, "Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software*, vol. 84, no. 3, pp. 397–414, mar 2011.

[397] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, dec 2007.

[398] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Gueheneuc, "A New Family of Software Anti-patterns: Linguistic Anti-patterns," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, mar 2013, pp. 187–196.

[399] S. L. Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The Effect of Lexicon Bad Smells on Concept Location in Source Code," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. IEEE, sep 2011, pp. 125–134.

[400] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building Empirical Support for Automated Code Smell Detection," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '10. New York, NY, USA: ACM, 2010, pp. 8:1—-8:10.

[401] V. Arnaoudova, M. Di Penta, and G. Antoniol, "Linguistic antipatterns: what they are and how developers perceive them," *Empirical Software Engineering*, vol. 21, no. 1, pp. 104–158, 2016.

[402] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

[403] D. Kirk, M. Roper, and N. Walkinshaw, "Using Attribute Slicing to Refactor Large Classes," in *Beyond Program Slicing*, ser. Dagstuhl Seminar Proceedings, D. W. Binkley, M. Harman, and J. Krinke, Eds., no. 05451. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum f{ü}r Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[404] B. Du Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman, "Does god class decomposition affect comprehensibility?" in *IASTED Conf. on Software Engineering*, 2006, pp. 346–355.

[405] M. Greiler, A. Van Deursen, and M.-A. Storey, "Automated Detection of Test Fixture Strategies and Smells," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, 2013, pp. 322–331.

[406] H. Neukirchen and M. Bisanz, "Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites," in *Proceedings of the 19th IFIP TC6/WG6.1 International Conference, and 7th International Conference on Testing of Software and Communicating Systems*, ser. TestCom'07/FATES'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 228–243.

[407] M. Greiler, A. Zaidman, A. van Deursen, and M.-A. Storey, "Strategies for avoiding text fixture smells during software evolution," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, may 2013, pp. 387–396.

[408] F. Khomh, S. Vaucher, Y.-G. Gueheneuc, and H. Sahraoui, "A Bayesian Approach for the Detection of Code and Design Smells," in *Quality Software, 2009. QSIC '09. 9th International Conference on*, 2009, pp. 305–314.

[409] M. Aniche, G. Bavota, C. Treude, A. V. Deursen, and M. A. Gerosa, "A validated set of smells in model-view-controller architectures," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 233–243.

[410] L. Punt, S. Visscher, and V. Zaytsev, "The a?b*a pattern: Undoing style in css and refactoring opportunities it presents," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Oct 2016, pp. 67–77.

[411] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 115–126.

[412] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, "An empirical study of code smells in javascript projects," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 294–305.

[413] W. Fenske, S. Schulze, D. Meyer, and G. Saake, "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)*, Sept 2015, pp. 171–180.

[414] D. Steidl and F. Deissenboeck, "How do java methods grow?" in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sept 2015, pp. 151–160.

[415] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 189–200.

[416] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA: ACM, 2016, pp. 59–69.

[417] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015.

Elder Vicente received his B.S.(2007) in control and automation engineering from the Polytechnic School of Uberlândia, Brazil. He earned his M.S.(2012) degree in Computer Science from the Federal University of Uberlândia. His research interests include software repository mining, software analytics and program comprehension.

Andrea De Lucia received the Laurea degree in computer science from the University of Salerno, Italy, in 1991, the MSc degree in computer science from the University of Durham, U.K., in 1996, and the PhD degree in electronic engineering and computer science from the University of Naples Federico II, Italy, in 1996. He is a Full Professor of software engineering at the Department of Computer Science of the University of Salerno, the Head of the Software Engineering Lab, and the Director of the International Summer School on Software Engineering. Previously, he was at the Department of Engineering and the Research Centre on Software Technology of the University of Sannio, Italy. His research interests include software maintenance and testing, reverse engineering and reengineering, source code analysis, code smell detection and refactoring, mining software repositories, defect prediction, empirical software engineering, search-based software engineering, traceability management, collaborative development, workflow and document management, and visual languages. He has published more than 250 papers on these topics in international journals, books, and conference proceedings and has edited books and journal special issues. He serves on the editorial boards of international journals and on the organizing and program committees of several international conferences in the field of software engineering. He is a senior member of the IEEE and IEEE Computer Society and was member-at-large of the executive committee of the IEEE Technical Council on Software Engineering.

Marcelo Maia is a Full Professor at the Faculty of Computing of the Federal University of Uberlândia (UFU), Brazil. He received his B.Sc. degree in Computer Science from Federal University of Uberlândia, in 1991. He received his M.Sc. (1994) and Ph.D. (1999) degrees in Computer Science from the Federal University of Minas Gerais (UFMG), Brazil. His interest area is Empirical Software Engineering and Programming Languages. He has published more than 80 peer-reviewed papers in journals and conferences. His current research interests include software analytics, software repository mining, program comprehension, and automated program repair.