A Review-based Comparative Study of Bad Smell Detection Tools

Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, Eduardo Figueiredo
Software Engineering Laboratory (LabSoft) – Department of Computer Science (DCC)
Federal University of Minas Gerais (UFMG) – Belo Horizonte – MG – Brazil
{eduardofernandes, johnatan-si, gustavovale, thpaiva, figueiredo}@dcc.ufmg.br

ABSTRACT
Bad smells are symptoms that something may be wrong in the system design or code. There are many bad smells defined in the literature and detecting them is far from trivial. Therefore, several tools have been proposed to automate bad smell detection aiming to improve software maintainability. However, we lack a detailed study for summarizing and comparing the wide range of available tools. In this paper, we first present the findings of a systematic literature review of bad smell detection tools. As results of this review, we found 84 tools; 29 of them available online for download. Altogether, these tools aim to detect 61 bad smells by relying on at least six different detection techniques. They also target different programming languages, such as Java, C, C++, and C#. Following up the systematic review, we present a comparative study of four detection tools with respect to two bad smells: Large Class and Long Method. This study relies on two software systems and three metrics for comparison: agreement, recall, and precision. Our findings support that tools provide redundant detection results for the same bad smell. Based on quantitative and qualitative data, we also discuss relevant usability issues and propose guidelines for developers of detection tools.

CCS Concepts
• General and reference → General literature • General and reference → Experimentation • Software and its engineering → Software maintenance tools • Software and its engineering → Software defect analysis.

Keywords
Systematic literature review; comparative study; bad smells; detection tools.

1. INTRODUCTION
Software maintenance and evolution are expensive activities and may represent up to 75% of the software development costs [39]. One reason for this fact is that the development efforts focus on addition of new functionality or bug correction rather than on design maintainability improvement [84]. Bad smells are an important factor affecting the quality and maintainability of a software system [22]. A bad smell is any symptom that may indicate a deeper quality problem in the system design or code [88]. For instance, we may consider duplicated code as a threat for future software maintenance tasks [82].

Bad smells can be detected in source code by either using manual or automated analyses [47]. Tools support automated analysis usually relying on different detection strategies, such as metric-based [70] and visualization support [49]. Since there are many bad smell detection tools proposed in the literature, it is hard to enumerate them and say what bad smells they are able to detect. Additionally, many tools are restricted to detect bad smells in specific programming languages. Therefore, by providing a coverage study, we can catalogue which smells are detected in each programming language, for instance.

Previous work [47][86] investigate the impact of bad smells on software quality and maintainability by studying the detection of smells in code. For instance, Fontana et al. [86] present a literature review covering seven bad smell detection tools and evaluate four of these tools in terms of their detection results. Similarly, Moha et al. [47] conduct an evaluation of a detection tool, called iPlasma [92], and compare it with other tools. The authors compute recall and precision for these tools. However, none of these studies provides an extensive overview of the research topic, as well as a comparison of bad smell detection tools.

Some developers may find it difficult to choose the most appropriate tool according to their needs. Moreover, some available tools may be using redundant strategies for detection of bad smells. In this context, this study provides a systematic literature review (SLR) of bad smell detection tools. For each identified tool, we show its features, such as its developed programming language, compatible languages for smell detection, supported types of bad smells, online availability for download, available documentation, and release year. In summary, we provide an overview of the state of the art with respect to tools for bad smell detection, aiming to identify trends, open challenges, and research opportunities.

As a result of our SLR, we found 84 bad smell detection tools proposed or used in research papers. These tools aim to detect 61 different bad smells by relying on at least six different detection techniques. They also target various programming languages, such as Java, C, C++, and C#. We discuss the most frequent bad smells these tools aim to detect, trying to track the interest of researchers in the detection of specific smells. From the 84 tools that we found, 29 of them are available online for download.

After the literature review, we conduct a comparative study with four available tools with respect to the detection of two Fowler's

EASE '16, June 01-03, 2016, Limerick, Ireland
 ISBN 978-1-4503-3691-8/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2915970.2915984

bad smells [88]: *Large Class* and *Long Method*. The studied tools are inFusion [19], JDeodorant [70], PMD [19], and JSpIRIT [72]. These four tools are free for use, designed to detect the two mentioned bad smells, and they we extracted from the 29 tools available online for download. The comparison of tools aims to assess agreement, recall, precision, and usability of the tools. As a result, we identify high agreement among tools with respect to the detection results, although JDeodorant points out more bad smell instances compared to the other tools. All four evaluated tools achieved poor precision rates (about 14%) in detecting Large Class. On the other hand, PMD and JSpIRIT performed well with respect to Long Method, achieving 50% to 67% of recall and 80% to 100% of precision.

The remainder of this paper is organized as follows. Section 2 describes the systematic literature review protocol we followed. Section 3 presents the results of the conducted literature review. Section 4 describes a comparative study of four bad smell detection tools for two selected bad smells. Section 5 discusses the lessons learned through this study. Section 6 presents limitations of this work. Section 7 discusses some related work. Finally, Section 8 concludes this study and points directions for future work.

2. LITERATURE REVIEW PROTOCOL

A Systematic Literature Review (SLR) is a study to provide identification, analysis and interpretation of evidences that relate to a particular research topic, supported by a protocol [91]. Three steps compose this SLR process: planning, conducting, and reporting. In the planning step, we identified the need of the literature review and, then, we defined the research questions and the review protocol. In the conduction step, we executed the defined protocol, covering from the initial selection of papers to the data extraction and analysis. Finally, we reported the obtained results for the target audience. This section describes these steps.

2.1 Goal and Research Questions

The goal of this systematic literature review is to identify and document all tools reported and used in the literature for bad smell detection. We defined this goal due to the wide number of proposed bad smell detection tools and the lack of a coverage study for summarizing them. We believe that tool developers and users would benefit of such a summarization and comparison of bad smell detection tools. In this context, we aim to investigate three research questions (RQs) described as follows.

RQ1: What are the bad smell detection tools proposed or used in literature papers?

RQ2: Which are the main features of these tools?

RQ3: Which are the most frequent types of bad smells these tools aim to detect?

To answer the first research question, RQ1, we perform: (i) an automated search in six electronic data sources and (ii) a manual filtering to the returned results. For RQ2, we choose to document the following features of each tool: name and release year, type of availability (i.e., plug-in, standalone, or both), user license, programming languages, supported bad smells and detection techniques, availability of documentation, and graphical user interface. Finally, by answering RQ3, we aim to identify all bad smells that researchers are interested in either by proposing detection tools or by investigating them in research studies.

2.2 Search String and Selection Criteria

There are various alternative terms for the concept of bad smell, such as design smell, code smell, and code anomaly. Since the goal of this study is to find the possible largest set of available tools for bad smell detection, we defined the following search string.

(tool AND ("bad smell*" OR "design smell*" OR "code smell*" OR "architecture smell*" OR "design anomaly*" OR "code anomaly*"))*

We used the "*" symbol in order to reach derived words from the previous prefix. For instance, the words *tools* and *tooling* can be included as a derivation from *tool**. Aiming to conduct searches in different electronic data sources, some adaptations were required to the presented search string. The search was applied on metadata only; i.e., title, abstract, and keywords. After pilot searching, we decided to exclude some general terms from the search string, such as design defect and design flaw, because they bring us many false positives.

The pilot searches also supported the definition of the scope of this study. For instance, by means of pilot searches, we defined the inclusion and exclusion criteria. Table 1 summarizes the inclusion and exclusion criteria we applied in this study. Aiming to restrict the papers included, we defined four inclusion criteria and three exclusion criteria. For instance, as inclusion criteria papers published in the Computer Science area and, as an example of exclusion criteria, papers should be at least two pages long.

Table 1. List of SLR Inclusion and Exclusion Criteria

Inclusion Criteria	Exclusion Criteria
Papers published in Computer Science	Papers published before 2000
Papers written in English	Papers shorter than two pages
Papers available in electronic format	Websites, leaflets, and grey literature
Propose or use bad smell detect tools	

We decided to include only papers published after 2000 in our study because of the publication of the Refactoring book by Fowler [88] in 1999. This book defines the most well-known bad smells. The pilot searches have also not returned any relevant paper before 2000. Therefore, our study began with the search for papers published since 2000 until 2015. However, after conducting the snowballing step, a search approach that uses citations in papers from a SLR as reference list to identify additional papers not covered by the SLR [99], we found relevant research papers published since 1993.

2.3 Electronic Data Sources

We run the defined search string in July 2015 in six electronic data sources: ACM Digital Library[1], IEEE Xplore[2], Science Direct[3], Scopus[4], Web of Science[5], and Engineering Village[6]. BibTeX and text files (in this case, converted to BibTeX) were

[1]http://dl.acm.org/
[2]http://ieeexplore.ieee.org/
[3]http://www.sciencedirect.com/
[4]http://www.scopus.com/
[5]http://webofknowledge.com/
[6]http://www.engineeringvillage.com/

imported to the JabRef[7] tool to manage the references. We manually download the BibTeX files of papers from ACM Digital Library, one by one, because it does not support automatic downloading. For the other electronic data sources, we were able to download the BibTeX or text file automatically.

Figure 1 illustrates the steps we followed for selecting papers with the search string executed in the six aforementioned electronic data sources. At first, we start with 1002 studies: 429 from ACM Digital Library (ACM), 65 from IEEE Xplore (IEEE), 10 from ScienceDirect (SD), 145 from Scopus (SC), 217 from Web of Science (WoS), and 136 from Engineering Village (EV).

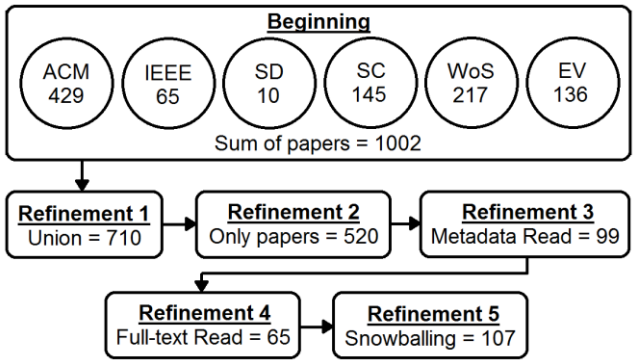

Figure 1. Steps for Selection of Papers

We then followed a five-step refinement process in order to select the final set of primary studies, as follows. Refinement 1 removed duplicated studies, resulting in 710 studies. Refinement 2 discarded non-papers, such as technical reports and conference calls. After this refinement, we obtained 520 papers. In Refinement 3, we read metadata from these 520 papers and kept only 99 studies in accordance with all inclusion and exclusion criteria. We then conducted a full-text paper read for these 99 papers, aiming to discard work that does not propose or use a bad smell detection tool, and kept 65 studies (Refinement 4). Finally, in Refinement 5, we conducted a snowballing procedure on the 65 papers, looking for citation of tools not covered by this set. After this step, we are able to retrieved 42 additional papers. Therefore, at the end, we obtained 107 studies according to the inclusion and exclusion criteria for data extraction. The papers used to extract data from tools are listed in our research group website [100] and their references (used in this paper) are from [1] to [80]. With respect to references, we prioritize papers that propose tools. When we were not able to find it, we relied on a paper that uses or cites a tool.

2.4 Data Extraction

For data extraction, we conducted a careful full-text read of the 107 selected primary papers. With respect to release year of tools, we considered the first identified value in a priority order. First, the integration year of an approach or algorithm for smell detection to an existing tool. If this data was not available, we considered the year of the paper that presented the tool. In case of not available information, we adopted the year of the first commercial off-the-shelf tool release. Finally, we considered the year of the earliest release of a tool documented on the Web.

[7]http://jabref.sourceforge.net/

By executing the SLR protocol, we found 84 bad smell detection tools. We recorded the extracted data about each tool in a spreadsheet for analysis, as well as relevant data from read papers that could be useful in writing this paper. We faced some difficulties in finding data related to some tools. For instance, some papers do not summarize the types of bad smells detected by the tool or they do not have a specific section to describe features of the tool (such as compatible language and applied detection technique). We tried to obtain as much information about the 84 tools as we could by searching on the Web.

2.5 Reporting

Since some bad smells have the same definition, but with different names, it was necessary to treat these cases for data analysis. We decided to consider the bad smells that shared a similar definition using a unique term, based on the Fowler's definition [88]. Table 2 lists the three bad smells that presented this naming clash problem. This table also shows the alternative names to the same bad smell. For instance, we considered Brain Method and God Method as alternative names of Long Method. The data analysis presented in Section 3 covers the entire set of 84 tools found. However, we considered only the 29 tools available online for download in our comparative study presented in Section 4.

Table 2. List of Bad Smells with Alternative Terms

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

3. RESULTS

This section presents the results of the systematic literature review. Section 3.1 reports an overview of the 84 tools we found in this study by release year. Sections 3.2 to 3.4 aim to answer the research questions RQ1 to RQ3 presented in Section 2.1.

3.1 Overview of Tools by Release Year

Figure 2 illustrates the number of bad smell detection tools by release year. We were not able to retrieve the release year of one tool, namely Analyst [18]. Therefore, we did not take into account this tool in Figure 2.

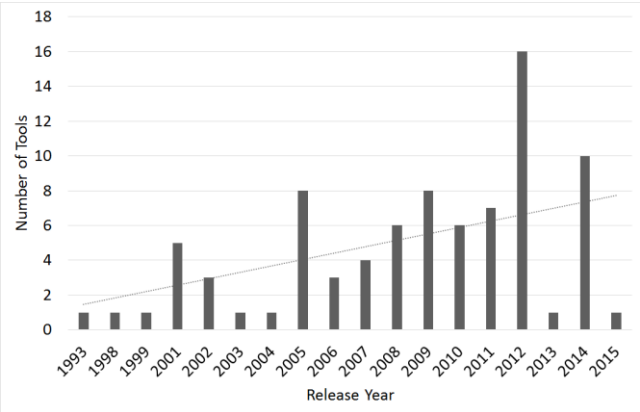

Figure 2. Number of Detection Tools Released by Year

Data in Figure 2 show a high number of bad smell detection tools released from 2005 to 2014. In fact, there is an increase tendency in the number of released tools, with a peak in 2012 with 16 released tools. Moreover, the highest number of proposed tools is in recent years, 2012 and 2014, although in 2013 it was released only one tool.

3.2 List of Bad Smell Detection Tools

The previous section shows that there is a high number of bad smell detection tools proposed and used in the literature. However, there is no study for summarizing them. Therefore, this section aims to answer the first research question (RQ1), defined as follows.

RQ1: What are the bad smell detection tools proposed or used in literature papers?

To answer this question, Table 3 presents the 84 bad smell detection tools we found in the literature review. In our study, we included tools to detect bad smells, even when the tool provides additional features, such as refactoring. We divided these tools in three categories: the first with the 29 tools that we were able to download and install (proposed or cited in literature); the second with the 54 tools proposed in literature but unavailable online for download; and the third group with one tool only cited in literature and not available for download.

Table 3. List of Bad Smell Detection Tools Found in this SLR

29 Tools Available Online for Download and Installation
Borland Together [77], CCFinder (CCFinderX) [29], Checkstyle [19], Clone Digger [8], Code Bad Smell Detector [22], Colligens [45], ConcernReCS [1], ConQAT [13], DECKARD [26], DuDe [75], Gendarme [53], inCode [77], inFusion [19], IntelliJ IDEA [17], iPlasma [43], Java Clone Detector (JCD) [28], jCosmo [71], JDeodorant [70], NiCad [10], NosePrints [53], PMD [19], PoSDef [9], SDMetrics [62], SpIRIT (JSpIRIT) [72], Stench Blossom [49], SYMake [67], TrueRefactor [20], Understand [65], Wrangler [37]
54 Tools Proposed in Literature but Unavailable Online
Absinthe [66], Anti-pattern Scanner [76], Arcoverde et al. [3], AutoMeD [78], Bad Smell Detection Tool (BSDT) [12], Bad Smells Finder [21], Bauhaus [59], Bayesian Detection Expert (BDTEX) [33], Bavota et al. [5], Baxter et al. [6], Bug Forecast [16], Clone Detector [64], CloneDetective [27], CocoViz [7], CodeSmellExplorer [57], CodeVizard [79], CP-Miner [38], Crespo et al. [11], Crocodile [63], DÉCOR [47], Dup [4], Duploc [14], EvoLens [58], Hamza et al. [23], Hayashi et al. [24], Hist-Inspect [42], iSPARQL [34], It's Your Code (IYC) [36], JCodeCanine [52], JSmell [61], Kaur and Singh [30], Keivanloo and Rilling [31], Kessentini et al. [32], Komondoor and Horwitz [35], Lui et al. [39], Matthew Munro [48], Mens et al. [46], Pradel et al. [56], PROblem DEtector O-O System (PRODEOOS) [44], Reclipse Tool Suite [73], Refactoring Browser [69], Ribeiro and Borba [60], SCOOP [40], Scorpio [25], Sextant [15], Smellchecker [55], SolidFX [68], Stasys Peldzius [54], SVMDetect [41], VCS-Analyzer [2], Wang et al. [74], WebScent [50], Xquery-based Analysis Framework (XAF) [51], Zang et al. [80]
1 Tool Cited but Unavailable Online for Download
Analyst [18]

In Table 3, a reference following each tool means either: a paper that proposes the tool (when it is available), a paper that uses the tool, a paper that cites the tool, or the tool's website. It is important to highlight that we only downloaded tools easily available online, that we found by searching on the Web or with an explicit link address in the selected paper. That is, we have not contacted authors or developers of tools to get access to a specific tool because we believe that actual users would not do this kind of direct contact.

3.3 Main Features of Detection Tools

This section presents and discusses the results of our systematic literature review with respect to the second research question (RQ2), defined as follows.

RQ2: Which are the main features of these tools?

By analyzing available data of the 84 tools, we observed that about 35.7% of the tools (30 in total) are plug-ins and 35.7% (30) are standalone applications. In addition, 4.7% (i.e., 4 tools) are available both as a plug-in and as a standalone application. These four tools are ConQAT [13], NosePrints [53], PMD [19], and SpIRIT (we call JSpIRIT from now on) [72]. We could not find information about the availability of 19 tools (i.e., 22.6%), despite of reading the tool papers and looking on the Web.

Furthermore, 35 bad smell detection tools (41.6%) have documentation available online. In this study, we considered website, tutorial, or research papers with a tool description as available documentation. Moreover, 60 tools (71.4%) provide a graphical user interface (GUI) and only four tools do not provide GUI. For the remaining 20 tools (23.8%), this information was unavailable.

Figure 3 shows the number of tools that aim to detect bad smells in some of the most popular programming languages. We presented data for nine of the top-ten most popular programming languages based on the IEEE Spectrum[8] ranking. Only for language R, the 6-th most used programming language worldwide, we did not find a bad smell detection tool. This ranking rely on data from different sources, such as GitHub, Google, and Stack Overflow. Although nine of the 10 most popular languages have at least one detection tool, there is a concentration of proposed tools for only three languages: Java, C, and C++. Moreover, languages such as PHP and JavaScript have few compatible tools. These findings point out to a research opportunity in less explored languages.

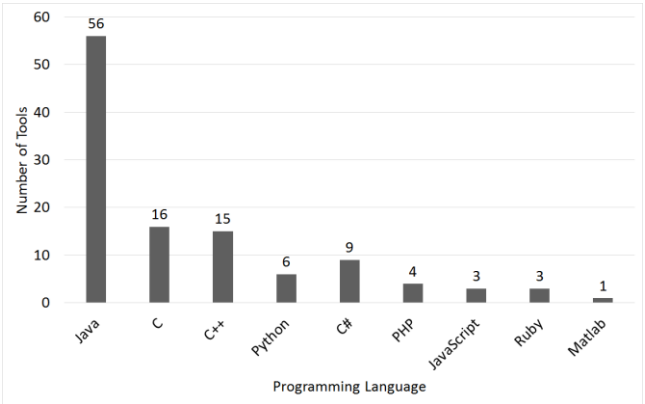

Figure 3. Programming Languages Tools Analyze

[8]http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2015

Figure 4 illustrates the programming languages that tools were developed. The ten listed languages were all languages for which we found implemented tools. Again, Java is predominant, followed by C and C++, in accordance with the top-ten languages from Spectrum. Moreover, some less popular languages appear in the list, such as Smalltalk and Erlang, with few developed tools.

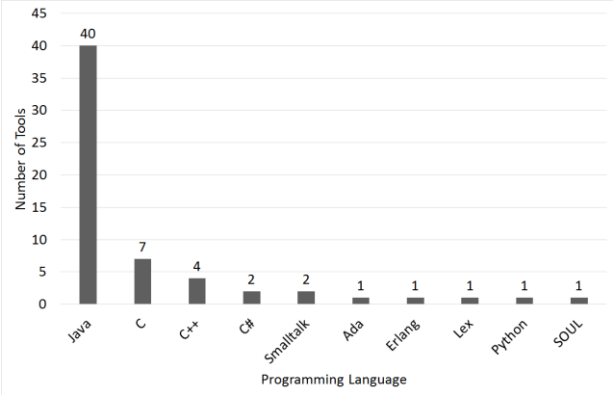

Figure 4. Programming Languages Tools Were Developed

With respect to the detection strategies, we found that 31 out of the 84 tools are metric-based (i.e., around 37% of the tools). For comparison, 15 (18%), 6 (7%), 5 (6%), and 3 (3.5%) of the tools are based on trees (such as AST), textual analysis, Program Dependence Graph (PDG), and token analysis, respectively. Moreover, 11 tools (approximately 13%) uses other strategies, such as machine learning and Logic Meta-programming (LMP). We were not able to find the detection technique applied by 17 of the 84 tools (around 20%). Note that the same tool may use combinations of detection techniques. Hence, the overall percentage may not be equal to 100%.

3.4 Detected Bad Smells

In addition to the features of selected tools presented in the previous section, another important feature of these tools is the list of detectable bad smells. Therefore, this section summarizes bad smells detected by each tool. In other words, we aim to answer the following research question.

RQ3: Which are the most frequent types of bad smells these tools aim to detect?

In total, this SLR found 61 different bad smells that tools can detect. From the 22 bad smells defined by Fowler [88], the found tools aim to detect 20 of them. The exceptions are *Alternative Classes with Different Interfaces* and *Incomplete Library Class*. In addition to 20 bad smells defined by Fowler, 41 smells defined by other authors [5][33][41][69][72] are detectable by the tools. These bad smells include *Dispersed Coupling* [72], *Functional Decomposition* [33], and *Spaghetti Code* [5], for instance. Figure 5 presents the top-ten most frequent bad smells that the found tools aim to detect. We can point out that all ten most recurrent smells are from Fowler's book.

Figure 5 also represents the percentage of tools that aim to detect each bad smell with respect to the entire set of 84 found tools. We highlight that *Duplicated Code* and *Large Class* are by far the major targets of detection tools. More than 40% of tools target at least one of these bad smells. One interesting question may emerge. Why are developers and researchers more interested in detecting these smells than others? The answer could be the high relevance of *Duplicated Code* to software engineering research. In the case of *Large Class*, we believe that it is target of many tools because it is probably one of the easiest bad smells to detect (in addition to its relevance to software engineering research).

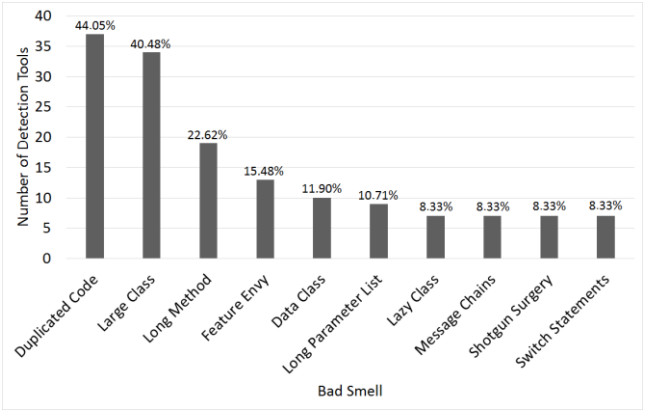

Figure 5. Top-ten Most Recurring Bad Smells

4. A COMPARATIVE STUDY

This section presents a comparative table of the 29 bad smell detection tools that we were able to download and install. We also performed a detailed comparative study of four tools, namely inFusion, JDeodorant, PMD, and JSpIRIT. Section 4.1 shows the comparison of the 29 tools and explains the criteria used to select the 4 tools for the comparative study. Section 4.2 presents the two bad smells (*Large Class* and *Long Method*) we choose to analyze in two software systems, namely JUnit [101] and MobileMedia [85]. Section 4.3 presents the overall number of bad smells detected by each tool. Based on these results, Section 4.4 reports and discusses the agreement among tools with respect to the detection of two bad smells in the selected systems. Finally, Section 4.5 verifies recall and precision of the chosen detection tools with respect to a reference list of bad smells.

4.1 Selection of Detection Tools

We found 84 bad smell detection tools in the systematic literature review reported in Sections 2 and 3. Since we could not get detailed information about all 84 tools, Table 4 presents general information of a selected set of 29 tools that we were able to download and install. In this table, "NA" means that data are not available (e.g., we were not able to find the data) and "AST" in the column Detection Technique means that the tool analyzes the Abstract Syntax Tree of a program. As far as information is available, we provide partial data about all 84 tools in the research group website [100].

We selected four bad smell detection tools for analysis in the comparative study, namely inFusion, JDeodorant, PMD, and JSpIRIT. The selection process was conducted as follows. First, we choose the Java programming language to study, since it is the most common language tools analyze (see Figure 3). We adopted similar criteria to select tools by their sets of bad smells, discussed in Section 4.2. We then restrict the set of tools to include only tools that are free for use, at least in a trial version. After applying these criteria, we end up with eight tools from the 29 listed in Table 4: Checkstyle, inFusion, iPlasma, JDeodorant, PMD, JSpIRIT, Stench Blossom, and TrueRefactor. However, Checkstyle, iPlasma, TrueRefactor, and Stench Blossom have been later discarded by different reasons, as discussed below.

In our study, we discarded Checkstyle because it was not able to detect any instance of the studied bad smells in the selected

Table 4. List of Bad Smell Detection Tools Available for Download

Tool Name	Plug-in	Detected Bad Smells	Language		Detection Technique	Free for Use	Guide	GUI	Release Year
			Developed	Detect					
Borland Together	Yes	Duplicated Code	Java	C#, C++, Java	Metrics	No	Yes	Yes	2011
CCFinder (CCFinderX)	No	Duplicated Code	C++	C, C#, C++, etc.	Token	Yes	Yes	Yes	2002
Checkstyle	Yes	Duplicated Code, Large Class, Long Method, Long Parameter List	Java	Java	NA	Yes	Yes	Yes	2001
Clone Digger	NA	Duplicated Code	Python	Java, Lua, Python	Tree	Yes	Yes	Yes	2008
Code Bad Smell Detector	No	Data Clumps, Switch Statements, and 3 other	Java	Java	AST	Yes	No	No	2014
Colligens	Yes	NA	C	C	NA	Yes	Yes	Yes	2014
ConcernReCS	Yes	Concern Smells: Primitive Concern Constant, and 5 other	Java	Java	Concern map	Yes	Yes	Yes	2012
ConQAT	Both	Clone Code	Java	ABAP, ADA, C++, C#, Java	Metrics	No	Yes	Yes	2005
DECKARD	No	Clone Code	C	Java	AST	Yes	Yes	No	2007
DuDe	No	Clone Code	Java	Language independent	Textual analysis	Yes	No	Yes	2005
Gendarme	No	Duplicated Code, Large Class, Long Method, and 4 other	C#	.NET, Mono	Rules	Yes	Yes	Yes	2006
inCode	Yes	Data Class, Data Clumps, Duplicated Code, and 2 other	Java	C, C++, Java	NA	No	Yes	Yes	2013
inFusion	No	Data Class, Data Clumps, Duplicated Code, and 2 other	NA	C, C++, Java	NA	No	Yes	Yes	2011
IntelliJ IDEA	No	Data Clumps, Feature Envy, Large Class, and 4 other	NA	Java, JavaScript, and 4 others	NA	No	Yes	Yes	2001
iPlasma	No	Duplicated Code, Feature Envy, Intensive Coupling, and 4 other	Java	C++, Java	Textual analysis	Yes	Yes	Yes	2005
Java Clone Detector	No	Duplicated Code	C++	Java	Tree	Yes	Yes	No	2009
jCosmo	No	InstanceOf, Switch Statement, Typecast	NA	Java	Tree	NA	Yes	Yes	2002
JDeodorant	Yes	Feature Envy, Large Class, Long Method	Java	Java	Metrics, AST	Yes	Yes	Yes	2007
JSpIRIT	Both	Data Class, Dispersed Coupling, Feature Envy, and 5 other	Java	C++, Java, Smalltalk	Metrics	Yes	Yes	Yes	2014
NiCad	Yes	Duplicated Code	C	C, C#, Java, etc.	NA	Yes	Yes	Yes	2011
NosePrints	Both	Feature Envy, Inappropriate Intimacy, Large Class, and 5 other	NA	NA	NA	No	No	Yes	2008
PMD	Both	Duplicated Code, Large Class, Long Method, Long Parameter List	Java	C, C#, C++, Java, PHP, and 11 other	NA	Yes	Yes	Yes	2008
PoSDef	Yes	NA	C#	UML diagrams	Metrics	Yes	No	Yes	2014
SDMetrics	No	Large Class	Java	UML diagrams	NA	No	Yes	Yes	2012
Stench Blossom	Yes	Comments, Data Clumps, and 4 other	Java	Java	Metrics	Yes	Yes	Yes	2010
SYMake	No	Cyclic Dependency, Duplicated Prerequisites	NA	C and Java	NA	Yes	Yes	Yes	2012
TrueRefactor	No	Lazy Class, Long Method, and 3 other	Java	Java	Graph	Yes	Yes	Yes	2011
Understand	No	NA	NA	C, C#, C++, etc.	NA	No	Yes	Yes	2008
Wrangler	Yes	Duplicated Code	Erlang	Erlang	Textual analysis	Yes	Yes	Yes	2010

software systems. We also discard iPlasma because we could not run the tool properly. In addition, we are aware that the same research group developed both iPlasma and inFusion. Therefore, these tools probably follow similar detection strategies. We have not used TrueRefactor in this comparative study because it does not provide an executable file in the package we downloaded. Finally, we also discarded Stench Blossom because it lacks a bad smell occurrence list (Stench Blossom is a visualization tool with no listing feature). Therefore, it is hard to validate their results, for instance, to calculate recall, precision, and agreement.

4.2 Selection of Bad Smells and Applications

As discussed in Section 3.4, the three most frequent bad smells detected by tools are *Duplicated Code*, *Large Class*, and *Long Method*. Since we are interested in investigating agreement among tools for the same bad smell, it seems natural for us to choose these three bad smells because most tools are able to detect them. However, we discard *Duplicated Code* because the nature of this bad smell makes it hard to quantity the aimed results: recall, precision, and agreement. In other words, we cannot easily compare the results of *Duplicated Code* neither among tools nor with the reference list because style of the output results vary a lot from one tool to another. Therefore, this comparative study focuses the analysis only on *Large Class* and *Long Method*.

For this comparative study, we selected two different software systems: JUnit version 4 [101] and MobileMedia version 9 (object-oriented version) [85]. JUnit is an open source Java testing framework and MobileMedia [85] is a software product line (SPL) for applications that manipulate photo, music, and video on mobile devices. Table 5 presents the number of classes, methods and lines of code of JUnit and MobileMedia. We choose these two software systems because they have been recurrently used in previous quality and maintainability-related studies [85][87][93][95]. JUnit is a well-known medium size open source project. We rely on JUnit to assess if tools are able to detect bad smells in a larger software system. Moreover, we have access to the MobileMedia developers and experts, and then we can recover a reference list of bad smells for MobileMedia.

With respect to the MobileMedia reference list protocol, we relied on two experts who used their own strategy for detecting, individually and manually, the bad smells in the classes and methods of the system. As a result, they returned two lists of entities. We merged these lists and discussed the findings with a

developer of the system to achieve a consensus and validate the entities that present a bad smell. The result of this discussion generated the final reference list used in this study. Table 6 presents the total number of *Large Classes* and *Long Methods* found in this system, according to the previously described protocol.

Table 5. Size metrics of JUnit and MobileMedia

Size Metrics	JUnit	MobileMedia
Number of Classes	983	55
Number of Methods	2948	290
Lines of Code (LOC)	26456	3216

Table 6. Reference list of bad smells in MobileMedia

Bad Smell	Occurrences
Large Class	7
Long Method	6
Total	13

4.3 Overall Results

For this comparative study, we configured two personal computers with two different operating systems. Two authors performed the same procedures, each one in a different computer. We decided to run our experiments in two different computers to assure the correctness of obtained data, through comparison of obtained results. Then, we aimed to minimize possible human mistakes in the retrieved results. We first installed the selected standalone tools in both computers, according to the tool version that is available for the specific operating systems. Then, we installed the plug-in tools in different Eclipse IDE instances. We rely on the default tool settings; i.e., no specific configuration was defined to any tool. Next, we used the detection tools to analyze the source code of the two Java applications: JUnit and MobileMedia.

After each tool has finished its execution, we recorded their detection results for the two bad smells studied in this study. Finally, we compared the results obtained in each personal computer to identify possible divergences. For all evaluated tools, both executions resulted in the same detected bad smell instances. Table 7 lists the number of detected *Large Classes (LC)* or *Long Methods (LM)* in JUnit and MobileMedia by each detection tool. Note that, in our study, we do not know the detection techniques applied by inFusion and PMD. On the other hand, we found that JSpIRIT uses metrics and JDeodorant uses both metrics and AST technique to detect bad smells. Considering that some tools apply unknown techniques, detection results may be different. We can observe that JDeodorant indicates the highest number of *Large Class* and *Long Method* instances in both software systems. In addition, apart from JDeodorant, tools have not detected *Long Method* instances in JUnit.

Table 7. Bad Smell Detection in JUnit and MobileMedia

Tool Name	Junit		MobileMedia	
	LC	LM	LC	LM
inFusion	0	0	1	2
JDeodorant	88	48	11	12
PMD	12	0	1	3
JSpIRIT	6	0	2	5

4.4 Agreement

To assess agreement among the selected detection tools, we compute the AC1 statistic coefficient [89]. It is a robust agreement coefficient alternative to the Cohen's kappa [83] and other common statistics for inter-rater agreement. It takes a value between 0 and 1, and reports the level of agreement using the following scale: Poor (< 0.20), Fair (0.21 to 0.40), Moderate (0.41 to 0.60), Good (0.61 to 0.80), and Very Good (0.81 to 1.00) [81]. Table 8 shows the agreement computed for the two analyzed applications, JUnit and MobileMedia, in terms of overall agreement (OA), AC1 coefficient (AC1), and 95% confidence interval (CI) of the detection tools. Note that, since Cohen's Kappa computes agreement between only two raters and we have multiple agreements (there are 4 tools to be compared), we decided to use a different agreement metric.

Regarding the JUnit analysis, we observed that the tools present a "Very Good" agreement in terms of *Large Class* detection results. However, for the same application, we were not able to compute agreement with respect to *Long Method* because only one tool (namely, JDeodorant) was able to identify this type of smell. Regarding the MobileMedia analysis, we conclude that the tools present a "Very Good" agreement in detecting both *Large Class* and *Long Method*.

In general, these results show the evaluated tools provide redundant detection results. However, we must consider the computed agreement may be high because the low number of bad smells in the applications. Therefore, a high agreement may indicate they agree with respect to non-detected elements instead of with detected bad smells.

4.5 Recall and Precision

To assess the accuracy of the studied tools to detect *Large Class* and *Long Method*, we computed recall and precision based on the bad smell reference list for the MobileMedia application. Table 9 presents the calculated values for each detection tool, with respect

Table 8. Agreement of Tools for JUnit and MobileMedia

	Large Class			Long Method		
Application	OA	AC1	95% CI	OA	AC1	95% CI
JUnit	88.55%	0.87	[0.84, 0.90]	-	-	-
MobileMedia	88.79%	0.87	[0.79, 0.94]	97.27%	0.97	[0.96, 0.98]

Table 9. Recall and Precision of the Tools for MobileMedia Only

Bad Smell	Recall				Precision			
	inFusion	JDeodorant	PMD	JSpIRIT	inFusion	JDeodorant	PMD	JSpIRIT
Large Class	14%	14%	14%	14%	100%	9%	100%	50%
Long Method	33%	33%	50%	67%	100%	17%	100%	80%

to MobileMedia only (for Junit, we have no reference list). With respect to recall, PMD and JSpIRIT provided the highest results. For instance, PMD and JSpIRIT achieved 50% and 67% of recall, respectively, when detecting *Long Method*. PMD is also the tool with the most accurate detection results for *Large Class* in the MobileMedia system. When analyzing precision, inFusion and PMD show the highest values. These tools always achieved 100% precision for both bad smells in the MobileMedia system. In general, JDeodorant scored the lowest results for both recall and precision. In turn, inFusion presented satisfactory results regarding precision, but low recall rates for both studied bad smells.

5. LESSONS LEARNED

In this section, we discuss some of the lessons learned in this study with respect to both the systematic literature review (Section 3) and the comparative study of detection tools (Section 4). Section 5.1 discuss some lessons with respect to the *Duplicated Code* bad smell. Section 5.2 summarizes the lessons about Large Class and Long Method detection. Finally, Section 5.3 presents usability issues we faced while installing and using the bad smell detection tools.

5.1 Duplicated Code

As pointed in Section 3.4, *Duplicated Code* is the most frequent bad smell that tools aim to detect. However, as discussed in Section 4.2, tools for detecting this smell present some usability issues that prevent us to quantitatively investigate recall, precision, and agreement of the tools. In fact, as discussed by Bellon et al. [82], *Duplicated Code* is a complex smell to detect. Therefore, we identify a research opportunity for tool developers to work on how to better present results of this bad smell in an easier way to be quantifiable and compared.

5.2 Large Class and Long Method

In the comparative study of tools with respect to *Large Class* and *Long Method* detection, we conclude that the proposed tools provide redundant detection results. We observed this redundancy by the high values of agreement for the studied tools that point to a "Very Good" agreement. Regarding recall, the tools presented low to medium rates for *Large Class* detection, indicating a possible open challenge. In respect to precision, two tools, namely inFusion and PMD, provided maximum precision (100%) that indicates to perfect accuracy of these tools in this case.

As discussed in Section 4.3, we were not able to know the detection techniques applied by inFusion and PMD. In turn, we observed that JDeodorant and JSpIRIT rely on metric-based detection strategies. Therefore, we may infer the redundancy may be due to similar detection techniques. In the context, by exploring alternative strategies than metrics, we may obtain different and, maybe, more effective results.

5.3 Usability Evaluation

By installing and using bad smell detection tools, we identified some usability issues about the tools that are worth to mention. Table 10 presents some features we observed in the four evaluated tools. For each tool, an "X" means that a tool supports a related feature. We can observe that inFusion is the only tool that supports all five features, although two of these are available only in the full and paid version of the tool.

With respect to the Result Export feature, for instance, we expected that results about the detected bad smells were easily exportable, for instance, to text, CSV or other file formats. However, some tools do not provide any way of exporting the results. In the set of the four tools evaluated in Section 4, only JDeodorant supports this feature and inFusion supports in a paid tool version. Another desirable feature is to highlight the smell-related source code. This feature makes it clear the source code related to a bad smell. All four detection tools somehow support this feature.

Table 10. Applicability Features of the Tools

Feature	inFusion	JDeodorant	PMD	JSpIRIT
Result export	X (in full version)	X		
Highlight smell occurrences	X	X	X	X
Allow detection settings	X	X	X	X
Graph visualization	X			
Detected Smell Filtering	X (in full version)	X	X	

In the case of metric-based detection strategies, tools usually provide the possibility of changing thresholds for metrics. Other tools also provide similar configuration options in order to set how sensible bad smell detection is in each specific context. All evaluated tools also support specific setting of the detection strategy. Finally, it is also desirable some enhanced visualization means to present the detection results using, for instance, graphical elements such as charts. Only inFusion provides such support. Apart from JSpIRIT, the evaluated tools support the Filtering feature . This feature allows hiding part of the results by setting visualization options.

In addition to the aforementioned features, we found that some usability issues could hinder the tool user experience. For instance, we faced some usability problems, such as difficulty to navigate between bad smell occurrences (in general, results are showed in long lists without summarization), difficulty to identify the source code related to a smell detection, and lack of advanced filters for specific bad smells we want to detect.

In general, we also observed that the tools do not provide data visualization through statistical analysis, counters of detection results, or results presentation by charts. These features could be useful in the comparison of tools. Furthermore, PMD and JSpIRIT do not provide way to export the results. This feature is helpful in data analysis and comparison. In this context, we suggest that developers of new detection tools should be aware of the possible usages of their tools, considering these observations.

6. THREATS TO VALIDITY

Even with the careful planning, different factors may affect these research results by invalidating its main findings. We discuss below strategies taken to reduce the impact of these factors on the validity of our research results. Some of these actions, mainly related to our research method, we adopted to increase the study confidence, aiming to reduce the threats to the study validity.

Scope and Strategy – For the systematic literature review, we selected six different electronic data sources, but there may be other sources with relevant papers. Nevertheless, we consider minimizing this threat by the use of data sources that aggregate papers from diversified publishers (for instance, Scopus and Web of Science). Furthermore, we attend to answer questions clearly and unambiguously, even if the answers provided are not the most conclusive.

Validation and Generalization of Data and Results – With respect to the review protocol, we designed a search string for restricting our research. This string includes more than 10 synonyms for "bad smell" and, so, we expect to have achieved a sufficient number of papers in the studied context. In addition, we performed a pilot search to define the terms to appear in the search string. However, we cannot assume that all existing related papers were included through this filtering.

Search String Execution – We run our search string in two different computers, and we downloaded the BibTeX and text files three times, obtaining identical results. Thereby, we expected to reduce errors in string run task with respect to human factors. In case of text files, we converted them to the BibTeX style of references. In case of ACM data source, we were not able to export references automatically and, then, we transcribed manually the main data about each found papers (author, title, year, and journal) in a BibTeX file.

To eliminate repeated papers and keep only papers (Refinements 1 and 2). One of us verified this condition manually and another audited the results. In Refinement 3 (read metadata and include papers), we performed three classification steps. In the first step, one author read each metadata paper and scored it (-2 for out of context paper, -1 for papers that seems to propose or use a tool without explicit use of search string words in metadata, 0 for doubt, 1 for papers that use a tool, and 2 for papers that propose a tool). In our study, papers scored by -2, -1, and 0 should be discarded. In the second step, another author repeated the previous procedure. This task returned another score for metadata. Finally, a third author decided to include or not a paper based on the scores given by the other two researchers. In case of doubt, the paper was included for the next step. We believe this protocol minimized biases by considering the point of view of three authors, and the agreement of at least two of them.

Full-text Analysis and Data Extraction – One of the authors was responsible to read fully the selected papers and extract proposed or used bad smell detection tools from them. Although the careful conduction of this, with no deadline for completion, we do not take other measure for risk reduction. Only an analysis performed by another author in 10% of randomly chosen data. In this context, some tools that detect bad smells may have been wrongly discarded. For instance, we discarded tools such as Archimetrix [98], FindBugs [90], and VisTra [97], because we considered that they are not related to bad smells detection according to our bad smell definition based on Fowler [88].

Cataloging Features of the Tools – We were not able to find the name of some tools. In these cases, we named a tool with the authors who proposed it. In turn, some tools were, in fact, an evolution of previous tools and, so, we named them with the name of the previous tool. Another issue in cataloging features was that some papers did not specify the detected bad smells. Therefore, we abstracted the maximum types of bad smells informed to compose the table of tools and features, available in the research group website [100].

Comparison of the Tools – Detection tools aim to identify bad smell in different ways, using diverse techniques and procedures. Moreover, some tools do not provide customization of detection mechanisms, such as thresholds for metrics. Therefore, we decided to use all the evaluated tools in their default configurations. However, other configuration settings would probably give different results.

7. RELATED WORK

To the best of our knowledge, we did not find work that performed a systematic literature review and an extensive evaluation of bad smell detection tools. In spite of that, we found some studies [47][86][94] that can be considered related to our research.

Fontana et al. [86] present a closely related study. They discussed the findings of a literature review (but, not systematic) covering seven detection tools, namely Checkstyle, DÉCOR, inFusion, iPlasma, JDeodorant, PMD, and Stench Blossom. Furthermore, they evaluated four tools, Checkstyle, inFusion, JDeodorant, and PMD, using six versions of a same software system as input. They concluded that the tools provide significantly different detection results for a same bad smell, and some results are redundant. In respect to agreement of tools, they found significant agreement results only for two bad smells: *Large Class* and *Long Parameter List*.

Another related work, by Moha et al. [47], evaluates detection tools, but without providing a systematic literature review. The authors present a comparative study of tools including a new one proposed by them, called iPlasma. By using a list of bad smells built through manual inspection of source code, they were able to compute recall and precision for iPlasma. However, their study does not compare an extensive set of detection tools, and there is no agreement computation.

We have previously conducted an *ad hoc* literature review of duplicated code detection tools [96]. In this previous study, we investigated the available tools for single software projects in the context of cross-project detection of *Duplicated Code*. In this review, we found 20 tools and conducted a comparative study among tools that were available online for download. In general, several usability issues may occur in the analyzed tools, such as the lack of an option to export results and problems related to usability of the tools. Finally, we proposed in this previous paper [96] some guidelines for future implementation of detection tools, considering our findings by studying these tools.

Unlike previous work, this paper provides an overview of the state of the art in bad smell detection tools through a systematic literature review. In addition, we present a comparative study of tools available online for download and compatible with two of the most frequent bad smells that tools aim to detect. We compute agreement, precision and recall of these tools. Furthermore, we discuss same usability issues of the bad smell detection tools identified through the comparative study.

8. CONCLUSION

Bad smells are symptoms of anomalies in source code that can indicate problems in a software system. Although we may conduct manual detections of bad smells, some tools support this activity, using different detection strategies and approaches. In this paper, we present the results of a systematic literature review on bad smell detection tools (Sections 2 and 3). We found a large set of 84 different tools, but only 29 of them are available online for download.

With respect to the 84 tools, we observe that the amount of standalone and plug-in tools are roughly the same. In addition, the review results show that Java, C, and C++ are the top-three most covered programming language for bad smell detection. Most of the 84 tools are implemented in Java and rely on metric-based detection technique. Finally, the review shows that *Duplicated*

Code, *Large Class*, and *Long Method* are the top-three bad smells that tools aim to detect.

In addition to the systematic literature review, we conducted a comparison of tools with respect to two of the most recurrent bad smells that tool are designed to detect (Section 4). Through this study, we observed that the analyzed tools provide redundant detection results, given the high agreement coefficient computed. The results of this comparative study indicate that JDeodorant is the tools that indicates more instances of bad smells in its default configuration. However, PMD achieved the most accurate detection results for Large Class, considering both recall and precision measurements. With respect to Long Method, the PMD and JSpIRIT tools provided better results. They achieved 50% and 67% of recall, respectively. All tools presented some usability issues discussed in the lessons learned (Section 5).

The main contributions of this paper can be summarized as follows. We first present a systematic literature review of bad smell detection tools that found 84 different tools, and catalogued them by relevant features, such as detected bad smells, programming language for detection of smells, and detection techniques. We also performed a comparative study of tools considering the most frequent bad smells that tools aim to detect (*Large Class* and *Long Method*). Based on the literature review and comparative study, we discuss quantitative (agreement, recall, and precision) and qualitative data (lessons learned) about the tools.

As future work, we suggest an investigation on the interest of developers with respect to detection of some specific bad smell, such as *Duplicated Code*, *Large Class,* and *Long Method*. Another suggestion for future work is to study the redundancy of detection tool results. In addition, developers of current and new detection tools should be consider the usability issues discussed in this paper and cover the less studied bad smells.

9. ACKNOWLEDGMENTS

This work was partially supported by CAPES, CNPq (grant 485907/2013-5), and FAPEMIG (grant PPM-00382-14).

10. REFERENCES

[1] Alves, P., Santana, D., and Figueiredo, E. ConcernReCS: Finding Code Smells in Software Aspectization. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pp. 1463-1464, 2012.

[2] Arcelli, F., Rolla, M., and Zanoni, M. VCS-analyzer for Software Evolution Empirical Analysis. In *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2014.

[3] Arcoverde, R., Macia, I., Garcia, A., and Von Staa, A. Automatically Detecting Architecturally-Relevant Code Anomalies. In *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pp. 90-91, 2012.

[4] Baker, B. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, pp. 49-49, 1993.

[5] Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., and Palomba, F. An Experimental Investigation on the Innate Relationship between Quality and Refactoring. *Journal of Systems and Software*, 2015.

[6] Baxter, I., Yahin, A., Moura, L., Anna, M., and Bier, L. Clone Detection using Abstract Syntax Trees. In *Proceedings of the 14th International Conference on Software Maintenance (ICSM)*, pp. 368-377, 1998.

[7] Boccuzzo, S. and Gall, H. Automated Comprehension Tasks in Software Exploration. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE)*, pp. 570-574, 2009.

[8] Bulychev, P. and Minea, M. Duplicate Code Detection Using Anti-unification. In *Proceedings of the 2nd Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE)*, 2008.

[9] Chaudron, M. R., Katumba, B., and Ran, X. Automated Prioritization of Metrics-Based Design Flaws in UML Class Diagrams. In *Proceedings of the 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 369-376, 2014.

[10] Cordy, J. and Roy, C. The NiCad Clone Detector. In *Proceedings of the 19th International Conference on Program Comprehension (ICPC)*, pp. 219-220, 2011.

[11] Crespo, Y., López, C., Marticorena, R., and Manso, E. Language Independent Metrics Support Towards Refactoring Inference. In *Proceedings of the 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE)*, pp. 18-29, 2005.

[12] Danphitsanuphan, P. and Suwantada, T. Code Smell Detecting Tool and Code Smell-Structure Bug Relationship. In *Spring Congress on Engineering and Technology (S-CET)*, pp. 1-5, 2012.

[13] Deissenboeck, F., Pizka, M., and Seifert, T. Tool Support for Continuous Quality Assessment. In *Proceedings of the 13th International Workshop on Software Technology and Engineering Practice (STEP)*, pp. 127-136, 2005.

[14] Ducasse, S., Rieger, M., and Demeyer, S. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the 15th International Conference on Software Maintenance (ICSM)*, pp. 109-118, 1999.

[15] Eichberg, M., Haupt, M., Mezini, M., and Schäfer, T. Comprehensive Software Understanding with SEXTANT. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pp. 315-324, 2005.

[16] Ferenc, R. Bug Forecast: A Method for Automatic Bug Prediction. In *Proceedings of the International Conference on Advanced Software Engineering and Its Applications (ASEA)*, pp. 283-295, 2010.

[17] Fontana, F., Mangiacavalli, M., Pochiero, D., and Zanoni, M. On Experimenting Refactoring Tools to Remove Code Smells. In *Proceedings of the Scientific Workshops on the 16th International Conference on Agile Software Development Proceedings of the (XP)*, 2015.

[18] Fontana, F., Mariani, E., Morniroli, A., Sormani, R., and Tonello, A. An Experience Report on Using Code Smells Detection Tools. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 450-457, 2011.

[19] Fontana, F., Zanoni, M., Marino, A., and Mantyla, M. Code Smell Detection: Towards a Machine Learning-based Approach. In *Proceedings of the 29th International Conference on Software Maintenance (ICSM)*, pp. 396-399, 2013.

[20] Griffith, I., Wahl, S., and Izurieta, C. TrueRefactor: An Automated Refactoring Tool to Improve Legacy System and Application Comprehensibility. In *Proceedings of the 24th International Conference on Computer Applications in Industry and Engineering (CAINE)*, 2011.

[21] Grigera, J., Garrido, A., and Rivero, J. A Tool for Detecting Bad Usability Smells in an Automatic Way. *Web Engineering*, pp. 490-493, 2014.

[22] Hall, T., Zhang, M., Bowes, D., and Sun, Y. Some Code Smells have a Significant but Small Effect on Faults. *Transactions on Software Engineering and Methodology (TOSEM)*, 2014.

[23] Hamza, H., Counsell, S., Loizou, G., and Hall, T. Code Smell Eradication and Associated Refactoring. In Proceedings of the *2nd Conference on European Computing Conference (ECC)*, 2008.

[24] Hayashi, S., Saeki, M., and Kurihara, M. Supporting Refactoring Activities using Histories of Program Modification. *IEICE Transactions on Information and Systems*, pp. 1403-1412, 2006.

[25] Higo, Y. and Kusumoto, S. Enhancing Quality of Code Clone Detection with Program Dependency Graph. In *Proceedings of the*

16th Working Conference on Reverse Engineering (WCRE), pp. 315-316, 2009.

[26] Jiang, L., Misherghi, G., Su, Z., and Glondu, S. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pp. 96-105, 2007.

[27] Juergens, E., Deissenboeck, F., and Hummel, B. CloneDetective – A Workbench for Clone Detection Research. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pp. 603-606, 2009.

[28] Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. Do Code Clones Matter?. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pp. 485-495, 2009.

[29] Kamiya, T., Kusumoto, S., and Inoue, K. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Sode. *Transactions on Software Engineering (TSE)*, pp. 654-670, 2002.

[30] Kaur, R. and Singh, S. Clone Detection in Software Source Code using Operational Similarity of Statements. *Software Engineering Notes (SEN)*, pp. 1-5, 2014.

[31] Keivanloo, I. and Rilling, J. Clone Detection Meets Semantic Web-based Transitive Closure Computation. In *Proceedings of the 1st International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, pp. 12-16, 2012.

[32] Kessentini, M., Mahaouachi, R., and Ghedira, K. What You Like in Design Use to Correct Bad-Smells. *Software Quality Journal*, pp. 551-571, 2013.

[33] Khomh, F., Vaucher, S., Guéhéneuc, Y., and Sahraoui, H. BDTEX: A GQM-based Bayesian Approach for the Detection of Antipatterns. *Journal of Systems and Software (JSS)*, pp. 559-572, 2011.

[34] Kiefer, C., Bernstein, A., and Tappolet, J. Mining Software Repositories with iSPAROL and a Software Evolution Ontology. In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR)*, 2007.

[35] Komondoor, R. and Horwitz, S. Tool Demonstration: Finding Duplicated Code using Program Dependences. *Lecture Notes in Computer Science*, pp. 383-386, 2001.

[36] Kreimer, J. Adaptive Detection of Design Flaws. *Electronic Notes in Theoretical Computer Science*, pp. 117-136, 2005.

[37] Li, H. and Thompson, S. Similar Code Detection and Elimination for Erlang Programs. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pp. 104-118, 2010.

[38] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. 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 (OSDI)*, pp. 289-302, 2004.

[39] Liu, H., Ma, Z., Shao, W., and Niu, Z. Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort. *Transactions on Software Engineering (TSE)*, pp. 220-235, 2012.

[40] Macia, I., Arcoverde, R., Cirilo, E., Garcia, A., and von Staa, A. Supporting the Identification of Architecturally-relevant Code Anomalies. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*, pp. 662-665, 2012.

[41] Maiga, A., Ali, N., Bhattacharya, N., Sabané, A., Guéhéneuc, Y. G., Antoniol, G., and Aimeur, E. Support Vector Machines for Anti-pattern Detection. In *Proceedings of the 27th International Conference on Automated Software Engineering (ASE)*, pp. 278-281, 2012.

[42] Mara, L., Honorato, G., Medeiros, F. D., Garcia, A., and Lucena, C. Hist-Inspect: A Tool for History-sensitive Detection of Code Smells. In *Proceedings of the 10th International Conference on Aspect-oriented Software Development (AOSD)*, pp. 65-66, 2011.

[43] Marinescu, C., Marinescu, R., Mihancea, P., and Wettel, R. iPlasma: An Integrated Platform for Quality Assessment of Object-oriented Design. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, 2005.

[44] Marinescu, R. Detecting Design Flaws via Metrics in Object-oriented Systems. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS)*, pp. 173-182, 2001.

[45] Medeiros, F. An Approach to Safely Evolve Program Families in C. In *Proceedings of the SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, pp. 25-27, 2014.

[46] Mens, T., Tourwé, T, and Muñoz, F. Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE)*, 2003.

[47] Moha, N., Gueheneuc, Y., Duchien, L., and Le Meur, A. DECOR: A Method for the Specification and Detection of Code and Design Smells. *Transactions on Software Engineering (TSE)*, pp. 20-36, 2010.

[48] Munro, M. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In *Proceedings of the 11th International Symposium on Software Metrics*, pp. 15-15, 2005.

[49] Murphy-Hill, E. and Black, A. An Interactive Ambient Visualization for Code Smells. In *Proceedings of the 5th Symposium on Software Visualization (SOFTVIS)*, pp. 5-14, 2010.

[50] Nguyen, H., Nguyen, H., Nguyen, T., Nguyen, A., and Nguyen, T. Detection of Embedded Code Smells in Dynamic Web Applications. In *Proceedings of the 27th International Conference on Automated Software Engineering (ASE)*, pp. 282-285, 2012.

[51] Nodler, J., Neukirchen, H., and Grabowski, J. A Flexible Framework for Quality Assurance of Software Artefacts with Applications to Java, UML, and TTCN-3 Test Specifications. In *Proceedings of the International Conference on Software Testing Verification and Validation (ICST)*, pp. 101-110, 2009.

[52] Nongpong, K. Integrating "Code Smells" Detection with Refactoring Tool Support. PhD thesis, University of Wisconsin-Milwaukee, 2012.

[53] Parnin, C., Görg, C., and Nnadi, O. A Catalogue of Lightweight Visualizations to Support Code Smell Inspection. In *Proceedings of the 4th Symposium on Software Visualization (SOFTVIS)*, pp. 77-86, 2008.

[54] Peldzius, S. Automatic Detection of Possible Refactorings. In *Proceedings of the 16th International Conference on Information and Software Technologies (ICIST)*, pp. 238-245, 2010.

[55] Pessoa, T., Monteiro, M., and Bryton, S. An Eclipse Plugin to Support Code Smells Detection. 2012.

[56] Pradel, M., Jaspan, C., Aldrich, J., and Gross, T. Statically Checking API Protocol Conformance with Mined Multi-Object Specifications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pp. 925-935, 2012.

[57] Raab, F. CodeSmellExplorer: Tangible Exploration of Code Smells and Refactorings. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 261-262, 2012.

[58] Ratzinger, J., Fischer, M., and Gall, H. EvoLens: Lens-view Visualizations of Evolution Data. In *Proceeding of the 8th International Workshop on Principles of Software Evolution (IWPSE)*, pp. 103-112, 2005.

[59] Raza, A., Vogel, G., and Plödereder, E. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In *Proceedings of the 11th International Conference on Reliable Software Technologies Reliable Software Technologies (Ada-Europe)*, pp. 71-82, 2006.

[60] Ribeiro, M. and Borba, P. Improving Guidance when Restructuring Variabilities in Software Product Lines. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 79-88, 2009.

[61] Roperia, N. JSmell: A Bad Smell Detection Tool for Java Systems. MSc dissertation, California State University, 2009.

[62] SDMetrics – The Software Design Metrics Tool for UML. http://www.sdmetrics.com (accessed January 15, 2016).

[63] Simon, F., Teinbruckner, F. S., and Lewerentz, C. Metrics Based Refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 30-38, 2001.

[64] Singh, S. and Kaur, R. Clone Detection in UML Class Models using Class Metrics. *Software Engineering Notes (SEN)*, pp. 1-3, 2014.

[65] Singh, V., Snipes, W., and Kraft, N. A Framework for Estimating Interest on Technical Debt by Monitoring Developer Activity Related to Code Comprehension. In *Proceedings of the 6th International Workshop on Managing Technical Debt (MTD)*, pp. 27-30, 2014.

[66] Stevens, R., De Roover, C., Noguera, C., Kellens, A., and Jonckers, V. A Logic Foundation for a General-Purpose History Querying Tool. *Science of Computer Programming*, pp. 107-120, 2014.

[67] Tamrawi, A., Nguyen, H., Nguyen, H., and Nguyen, T. SYMake: A Build Code Analysis and Refactoring Tool for Makefiles. In *Proceedings of the 27th International Conference on Automated Software Engineering (ASE)*, pp. 366-369, 2012.

[68] Telea, A., Byelas, H., and Voinea, L. A Framework for Reverse Engineering Large C++ Code Bases. *Electronic Notes in Theoretical Computer Science*, pp. 143-159, 2009.

[69] Tourwé, T. and Mens, T. Identifying Refactoring Opportunities using Logic Meta Programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 91-100, 2003.

[70] Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. JDeodorant: Identification and Removal of Type-checking Bad Smells. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 329-331, 2008.

[71] Van Emden, E., and Moonen, L. Java Quality Assurance by Detecting Code Smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, pp. 97-106, 2002.

[72] Vidal, S., Marcos, C., and Díaz-Pace, J. An Approach to Prioritize Code Smells for Refactoring. *Automated Software Engineering*, pp. 1-32, 2014.

[73] Von Detten, M., Meyer, M., and Travkin, D. Reverse Engineering with the Reclipse Tool Suite. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pp. 299-300, 2010.

[74] Wang, C., Hirasawa, S., Takizawa, H., and Kobayashi, H. A Platform-Specific Code Smell Alert System for High Performance Computing Applications. In *Proceedings of the 28th International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pp. 652-661, 2014.

[75] Wettel, R. and Marinescu, R. Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Fragments. In *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2005.

[76] Wieman, R. Anti-pattern Scanner: An Approach to Detect Anti-patterns and Design Violations. Doctoral dissertation, Delft University of Technology, 2011.

[77] Yamashita, A. and Moonen, L. To What Extent Can Maintenance Problems be Predicted by Code Smell Detection? – An Empirical Study. *Information and Software Technology*, pp. 2223-2242, 2013.

[78] Yang, L., Liu, H., and Niu, Z. Identifying Fragments to be Extracted from Long Methods. In *Proceedings of the 16th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 43-49, 2009.

[79] Zazworka, N. and Ackermann, C. CodeVizard: A Tool to Aid the Analysis of Software Evolution. In *Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2010.

[80] Zhang, L., Sun, Y., Song, H., Wang, W., and Huang, G. Detecting Anti-patterns in Java EE Runtime System Model. In *Proceedings of the 4th Asia-Pacific Symposium on Internetware*, pp. 21, 2012.

[81] Altman, D. Practical Statistics for Medical Research. Chapman & Hall, London, 1991.

[82] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering (TSE)*, pp. 577-591, 2007.

[83] Cohen, J. A Coefficient of Agreement for Nominal Scales. *Educational and Psychosocial Measurement,* 20, pp. 37-46, 1960.

[84] Dig, D., Manzoor, K., Johnson, R., and Nguyen, T. Refactoring-Aware Configuration Management for Object-Oriented Programs. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pp. 427-436, 2007.

[85] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan1, S., Castor Filho, F., Dantas, F. Evolving Software Product Lines with Aspects. In *Proceedings of 30th International Conference on Software Engineering (ICSE)*, pp. 261-270, 2008.

[86] Fontana, F., Braione, P., and Zanoni, M. Automatic Detection of Bad Smells in Code: An Experimental Assessment. *Journal of Object Technology*, 2012.

[87] Fontana, F., Rolla, M., and Zanoni, M. Capturing Software Evolution and Change through Code Repository Smells. In *Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation*, pp. 148-165. Springer International Publishing, 2014.

[88] Fowler, M. Refactoring: Improving the Design of Existng Code. Object Techonology Series. Addison-Wesley, 1999.

[89] Gwet, K. Handbook of Inter-Rater Reliability: The Definitive Guide to Measuring the Extent of Agreement among Raters. Advanced Analytics, LLC, 2014.

[90] Hovemeyer, D. and Pugh, W. Finding Bugs is Easy. *ACM Sigplan Notices*, pp. 92-106, 2004.

[91] Kitchenham B., Charters, S. Guidelines for Performing Systematic Literature Reviews in Software Engineering. In Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

[92] Lanza, M. and Marinescu, R. Object-oriented Metrics in Practice: using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems. Springer Science & Business Media, 2007.

[93] Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., and von Staa, A. 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 (AOSD)*, pp. 167-178, 2012.

[94] Mens, T. and Tourwé, T. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering (TSE)*, pp. 126-139, 2004.

[95] Murphy-Hill, E., Parnin, C., and Black, A. How We Refactor, and How We Know It. *Transactions on Software Engineering (TSE)*, 2012.

[96] Oliveira, J., Fernandes, E., and Figueiredo, E. Evaluation of Duplicaded Code Detection Tools in Cross-project Context. In *Proceedings of the 3rd Workshop on Software Visualization, Evolution, and Maintenance (VEM)*, pp. 49-56, 2015.

[97] Štolc, M. and Polášek, I. A Visual Based Framework for the Model Refactoring Techniques. *In Proceedings of 8th International Symposium on Applied Machine Intelligence and Informatics (SAMI)*, 2010.

[98] Von Detten, M. Archimetrix: A Tool for Deficiency-Aware Software Architecture Reconstruction. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*, pp. 503-504, 2012.

[99] Wohlin, C. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2014.

[100] Fernandes, E., Oliveira J., Vale, G., Paiva, T., Figueiredo, E. A Review-based Comparative Study of Bad Smell Detection Tools: Data of the Study. Software Engineering Laboratory (LabSoft). http://goo.gl/qMbRus (accessed November 5, 2015).

[101] JUnit – A Programmer-oriented Testing Framework for Java. https://github.com/junit-team/junit (accessed November 6, 2015).