Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at *Information* **2018**, *9*, 273; doi:10.3390/info9110273

Article

The Impact of Code Smells on Software Bugs: a Systematic Literature Review

Aloisio Sampaio Cairo [1], Glauco de Figueiredo Carneiro [1,*], Miguel Pessoa Monteiro [2]

[1] Programa de Pós-Graduação em Sistemas e Computação (PPGCOMP), Universidade Salvador (UNIFACS), Salvador 41770-235, Brazil; aloisiocairo@gmail.com

[2] NOVA LINCS, Faculdade de Ciências e Tecnologia da Universidade NOVA de Lisboa (FCT/UNL), 2829-516 Caparica, Portugal; mtpm@fct.unl.pt

* Correspondence: glauco.carneiro@unifacs.br; Tel.:+55-71-3330-4630

Abstract: *Context:* Code smells are associated with poor design and programming style that often degrades code quality and hampers code comprehensibility and maintainability. *Goal:* Identify reports from the literature that provide evidence of the influence of code smells on the occurrence of software bugs. *Method:* We conducted a Systematic Literature Review (SLR) to reach the stated goal. *Results:* The SLR includes selected studies from July 2007 to September 2017 which analyzed the source code for open source and proprietary projects, as well, as several code smells and anti-patterns. The results of this SLR show that 24 code smells are more influential in the occurrence of bugs according to 16 studies. In contrast, three studies reported that at least 6 code smells are less influential in such occurrences. Evidence from the selected studies also point out tools, techniques and procedures applied to analyze the influence. *Conclusion:* To the best of our knowledge, this is the first SLR to target this goal. This study provides an up-to-date and structured understanding of the influence of code smells on the occurrence of software bugs based on findings systematically collected from a list of relevant references in the latest decade.

Keywords: code smells; code fault-proneness; bugs; software evolution

---

1. Introduction

Increased maintenance costs often are a consequence of poor software design, bad coding style and undisciplined practices. Error proneness is a frequent result, among other factors [1]. *Code smells* [2] are symptoms in source code that are indicative of bad style and/or design. They violate good practices and design principles, affecting understanding, maintenance and promoting a negative impact on software quality and ease of evolution [3].

Code smells have been largely discussed by the both the software engineering community and practitioners from the industry. There are several tools to support the detection of code smells in programs written in different languages [4][1,3–13]. Similarly, there are plenty of tools and also to report and manage issues reported as bugs in a software project. However, to the best of our knowledge, there are very few solutions that integrate these two activities and delve deeper on the ways code smells influence the occurrence of bugs. Regarding the detection of code smells, available tools usually adopt one of the six following techniques: manual, symptom-based detection, metric-based, probabilistic, search-based and cooperative-based approaches [14][15][16]. Many techniques use software metrics to detect code smells, some of which are metrics extracted from third-party tools and subsequently applied threshold values. In contexts of intensive use, this approach can be attractive for automatic detection. Though many studies discuss code smells, just a relatively small number focus on the analysis of how code smells influence bugs arising in software projects. The results of the systematic literature review reported in this article indicate that only one study discussed the use of a tool aimed at showing such a relationship in a given project [13].

In this article, we review published studies by means of a systematic literature review (SLR). To our knowledge, this is the first systematic review to present evidence from studies that analyzed the

© 2018 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at Information 2018, 9, 273; doi:10.3390/info9110273

influence of code smells on the occurrence of bugs in software projects. From the 18 selected studies, 16 studies reported the influence of code smells on the occurrence of bugs. Analyzing these 16 studies, we identified 24 code smells with greater influence on bugs, specially the God Class, Comments, Message Chain and Feature Envy code smells. It was also possible to identify the tools, techniques and resources that were used by each study. It was identified the use of 25 tools, 9 tools for the detection of code smells, and the other 16 tools, were used in some stage of the study, allowing access to the necessary information for the analyzes. It was also identified, the adoption of 24 techniques and resources, as for example: metric, heuristics, scripts, etc .

The rest of this article is organized as follows. Section 2 presents the background related to code smells and software bugs. Section 3 presents the problem statement/scope and emphasizes the differences between this Systematic Literature Review (SLR) and others that focus on related topics. Section 4 outlines the research methodology. Section 5 discusses the results of the SLR and characteristics of the selected works. Section 7 concludes and presents plans for future work.

2. Code Smells and Software Bugs

Martin Fowler coined the term *code smells* illustrated in a catalog of 22 symptoms in code to be used as indicators of poor design and implementation choices taken by developers [2]. In other words, code smells are *symptoms* in the source code indicating that there may be deficiencies in the design and/or programming style of a software system [2]. Depending on the scenario, a code smell may affect a class, a single method, a group of these or an entire subsystem. Further work extended and performed adjustments to the initial catalog of smells, leading to new versions of code smells, sometimes with new names [17–22]. This situation can be illustrated by the *God Class* and *Brain Class* smells proposed in [22] and derived from the *Large Class* smell originally proposed by Fowler [2]. During the analysis of the selected studies to answer the research questions, we found out that some authors use alternative names for the code smells, i.e., not following the name originally coined in Fowler's catalog. In table 1, we represent Fowler's original name along with corresponding alternative names.

Code Smells (Fowler [2])	Alternative Name
Duplicated Code	*Clones (S15), Code clones (S1, S17)*
Comments	*Comments Line (S12)*
Long Method	*God Method (S4)*
Long Parameter List	*Long Parameter List Class (S5)*

Table 1. List of Code Smells according to Fowler [2] and Corresponding Alternative Names

Many authors associate the occurrence of smells with poor quality [5,9,10,13,14,22–26], which hampers reusability, and maintainability. Software systems that have such symptoms are prone to develop bugs over time, increasing risks in the maintenance activities and therefore contributing to a troublesome and costly maintenance process. Such situations often translate into higher fault rates and more bugs filed by users and developers. Esses problemas são mais evidentes pelo fato developers are often unaware of code smells, como nos resultados apresentados em [27], que found that a considerable portion (32%) of developers did not know about code smells.

According to the second Lehman's law [28], as the project grows, there is a tendency to increase complexity as well. This is a context in which code smells find favorable conditions to arise in the evolving code. Studies report that code smells appear during change activities and when software bugs are fixed[29]. To tackle this problem, approaches and tools were implemented to support the identification of code smells [14–16,18,21,30–35]. According to [36], agile methods adopted refactoring techniques to eliminate code smells and hence reduce development and maintenance related costs.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at *Information* **2018**, *9*, 273; doi:10.3390/info9110273

3. Problem Statement and Scope

Code smells have been target by researchers as a relevant theme in software engineering [21,37,38]. According to [5], classes betraying code smell are more likely to contain bugs compared to classes devoid of them [5]. In one study [10] 53% of the identified problems are not directly related to code as follows: lack of adequate technical infrastructure, developer's coding habits, external services (e.g., web services), run-time environment (e.g., JRE), and defects initially present in the system. However, in the remaining, from the 47% associated with Java source code, up to 27% was related to code smells.

We conducted a systematic literature review with the purpose of identifying studies that investigate the influence of code smells in the occurrence of software bugs. To the best of our knowledge, this is the first SLR focusing on this theme, covering studies that analysed both open source and proprietary projects, as well as several types of code smells and anti-patterns that have cause-effect relationship with software bugs.

4. Research Methodology

This section describes the methodology applied for the phases of planning, conducting and reporting the review. In doing so, it also characterizes this work's scope.

In contrast to a non-structured review process, a Systematic Literature Review (SLR) [39], reduces bias and follows a precise and rigorous sequence of methodological steps to review the scoped literature. SLRs rely on well-defined and evaluated review protocols to extract, analyse, and document results as the steps conveyed in Figure 1.

Identify the needs for an SLR. Search for evidence in the literature of how code smells influence the occurrence of software bugs and which tools and/or resources practitioners and researchers have adopted to identify evidence of this influence.

Specifying the research questions. We aim to answer the following research questions by conducting a methodological review of existing research: **Research Question 1 (RQ1)**: *To which extent code smells influence the occurrence of software bugs?* Knowledge of the influence that code smells exert on bugs can assist practitioners and researchers in identifying bugs over releases and also in how to effectively plan their mitigation and/or fixing. **Research Question 2 (RQ2)**: *Which tools, resources and techniques have been adopted to identify evidence of the influence of code smells on the occurrence of software bugs?* Knowledge of which tools and resources have been used to identify evidence of this influence is an opportunity to motivate practitioners and researchers to use them in their projects.

4.1. Searching for Study Sources

A selection of keywords was made for the search for primary study sources, based on the above research questions. We searched for studies published in journals and conferences from July 2007 to September 2017. The search string below was applied to three digital libraries meeting the guidelines described in [39] on the basis of three points of view: population, intervention and result.

Search String: *("open source software" OR "OSS" OR "open source") AND ("code-smell" OR "bad smell" OR "code anomalies") AND ("bug" OR "failure" OR "issues" OR "fault" OR "error")*

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at Information 2018, 9, 273; doi:10.3390/info9110273

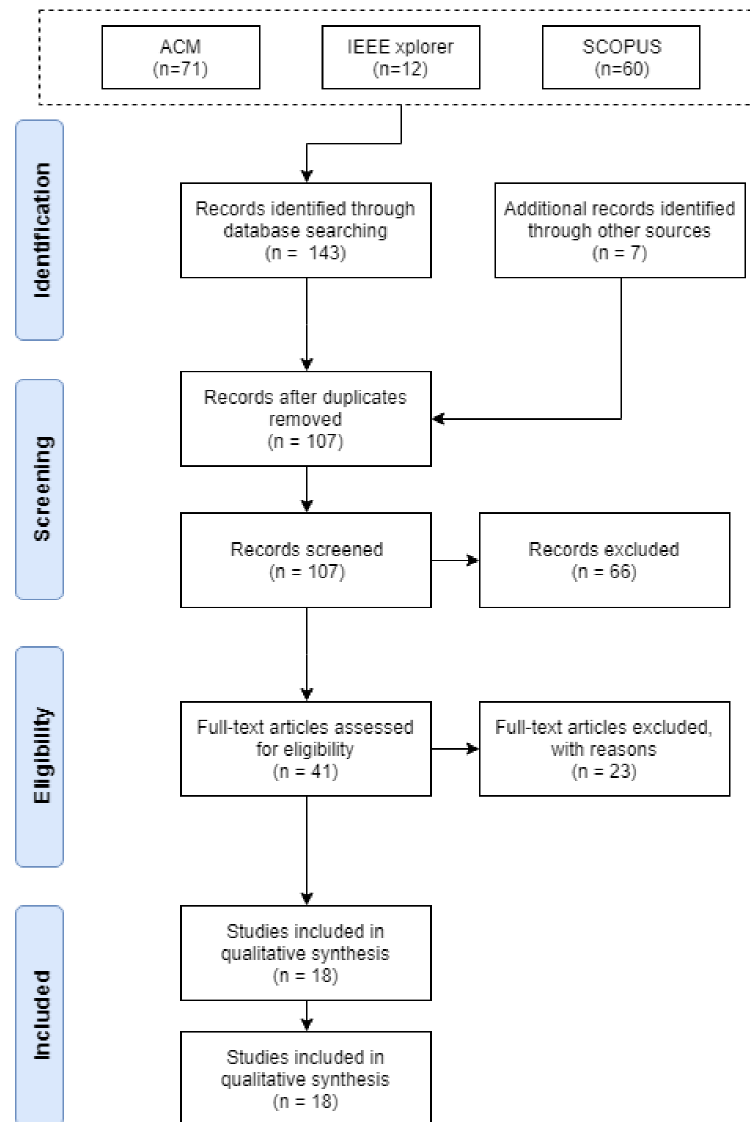

Figure 1. Stages of the study selection process

4.2. Selection of Primary Studies

To carry out the selection of studies, the following steps were carried out.

Step 1 - Search string results applied in digital libraries: Three digital libraries were selected: IEEE Xplore Digital Library, ACM Digital Library and SCOPUS. They were selected due to their relevance in Software Engineering [40]. The syntax of the search string was the same for the three libraries, including just the title, keywords and abstract. Only SCOPUS required some slight adaptations to its specific search model, though preserving the terms. The automatic search was complemented by a manual selection of works related to the research question. Duplicated reports were discarded.

Stage 2—Read titles and abstracts to identify potentially relevant studies—Identification of potentially relevant studies, based on the analysis of title and abstract, discarding studies that are clearly irrelevant to the search. When there was some doubt about whether a study should be included or not, it was included for consideration at later stages.

Step 3 – Apply inclusion and exclusion criteria: after the previous steps, the introduction, the methods and the conclusion were read and the inclusion and exclusion criteria were applied. In case of doubt about classification, a more thorough and complete reading was made.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at *Information* **2018**, *9*, 273; doi:10.3390/info9110273

Inclusion criteria. IC1: Publications should be "journal" or "conference". **IC2:** Works that involve some empirical study or present "lessons learned" (e.g., experience report). **IC3:** If several journal articles report the same study, the most recent article should be included.

Exclusion criteria. EC1: Studies that do not answer the research question. **EC2:** Studies merely based on expert advice without providing solid evidence. **EC3:** Publications which are previous versions of work also presented in later publications. **EC4:** Publications published before July 2006 *(note: publications after January 2018 are not included in this SLR due to the fact that this was the month in which the collection was conducted)*. **EC5:** Studies that are focused only on code smells or on software bugs without any discussion regarding the relationship and/or influence that the first exert on the second.

Step 4 – Critical evaluation of the primary studies at hand: After completing the previous steps, the primary studies at hand were submitted to the quality criteria suggested by Dybå et al. [41].

Quality Criteria. QC1: Is the document based on empirical evidence or just based on expert opinion? **QC2:** Is there a clear statement of the research objectives? **QC3:** Is there an adequate description of the context in which the research was carried out? **QC4:** Was the research project suitable to address the research objectives? **QC5:** Was data collected to address the research question? **QC6:** Is there a clear statement of the findings?

4.3. Data Extraction

Based on the results of the selection process described in section 4.2, we listed and classified the selected primary studies in a spreadsheet to enable a clear understanding of aims, methodologies and findings. The spreadsheet organized the selected studies using the following fields: (i) identification number; (ii) year; (iii) title; (iv) objectives or aims; (v) code smells; (vi) programming language; (vii) analysed projects; (viii) solution to establish a relationship among bugs and project's commits; (ix) research questions and respective answers; (x) code smells that have influence on bugs; (xi) software projects that illustrate this influence.

4.4. The Automatic Search

We performed an automatic search targeting three digital libraries based on a sequence of relevant keywords derived from the research questions. We started the review with an automatic search, followed by a manual search, to identify potentially relevant studies and afterwards applied the inclusion/exclusion criteria. The first tests using automatic search began in January 2018. We had to adapt the search string in some of the repositories presented in Table 2, taking care to preserve its primary meaning and scope. The manual search consisted of studies published in conference proceedings and journals that were included by the authors while searching the theme in different sources. These studies were likewise analysed regarding their titles and abstracts and were represented in a spreadsheet to facilitate the next step, identifying the potentially relevant studies. Figure 1 conveys them as 7 studies which, compared to the 143 studies selected through the automatic search, represent approximately 4,89% of all papers. Thus, the automatic search provided the majority of the analysed papers of this SLR. We tabulated everything on a spreadsheet so as to facilitate the subsequent phase of identifying potentially relevant studies. Figure 1 and Table 2 present the results obtained from each electronic database used in the search, which resulted in 143 articles spanning all databases.

4.5. Potentially Relevant Studies

Results obtained from both the automatic and manual search were included on a single spreadsheet. Articles with identical title, author(s), year and abstract were considered duplicates and thus discarded. As shown in figure 1, the search (Step 1) returned a total of 143 references in the automatic search and 7 from the separated manual search (*Step 1*), totaling 150 studies. After reading title and abstracts, we considered 107 papers as relevant (Step 2).

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at Information 2018, 9, 273; doi:10.3390/info9110273

At *Step 3*, we read the the methodology section, the research questions and corresponding results of 18 articles considered as relevant for the research questions. Finally, at (*Stage 4*), answers to the two research questions—**RQ1** and **RQ2**—were obtained. Table 2 presents an overview of the selection process per public data source.

Along with the automatic survey, snowballing was conducted in which it located a considerable amount of studies. It does does not invalidate the string of SLR, as snowballing can be used in systematic reviews, as described in a previous study [42] that performed comparison between a database search and the snowballing technique, which yielded the same references. Snowballing was performed in the articles obtained through the automatic search and selected to answer the research question. These articles are available in table A2 where the third column provides the source of each article.

Table 2 presents an overview of the process selection per public data source.

Public Data Source	Search Result	Relevant Studies	Search Effectiveness
ACM	71	4	5,63%
IEEE	12	2	16,6%
SCOPUS	60	5	8,33%

Table 2. Selection process per public data source

5. Results and Discussions

This section presents the SLR results to answer research questions **RQ1/RQ2** and characteristics of the selected studies. The selected studies are all listed in Tables A1 and A2 and identified with the letter "S", followed by the number of the study.

Figure 2 depicts the temporal distribution of the selected studies. It should be noted that from the 18 studies, 6 of them were published in 2017. To the best of our knowledge, this is the first systematic literature review to analyse studies targeting on the impact of code smells on software bugs.

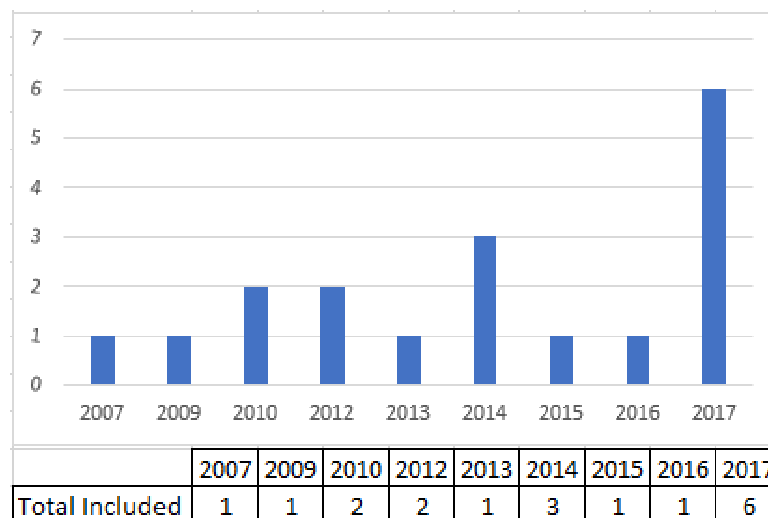

Figure 2. Timeline distribution of papers

5.1. Influence of Code Smells on Software Bugs

In this subsection, we use evidence collected from the selected studies to answer **Research Question (RQ1):** *To which extent code smells influence the occurrence of software bugs?*

We present in Table 3 how each of the selected studies answer **RQ1**. In this case, the following studies provide evidence of the influence of code smells in the occurrence of software bugs: S1, S2, S3, S4, S5, S6, S7, S8, S9, S11, S12, S13, S14, S16, S17, S18. On the other hand, studies S10 and S15 advised

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at Information 2018, 9, 273; doi:10.3390/info9110273

that based on the collected evidence there is no cause and effect relationship between code smells and the occurrence of software bugs. In other words, the literature has pointed out that the majority of studies indicate that classes affected by code smells tend to be more change- and fault-prone than the others not affected. In the following paragraphs of this subsection, we discuss evidence that support this result.

Do code smells influence the occurrence of software bugs?	Selected Studies
Yes	S1, S2, S3, S4, S5, S6, S7, S8, S9, S11, S12, S13, S14, S16, S17, S18
No	S10, S15

Table 3. Influence of the code smell on bugs

Figure 3. Influential Code Smells on Software Bugs

The Figure 3 classifies code smells according to their influence on software bugs based on evidence provided by the selected studies. These studies correspond to those whose answer was *Yes* in Table 3. Therefore, studies S10 and S15 were not considered as sources for this classification. The *Not informed* node shown in figure 3, is not a type of code smell. This node indicates that the studies related to it did not indicate which code smells had a greater or lesser influence on the bugs of the projects analyzed in each study.

In S1 [5], the authors argued that class fault proneness is significantly higher in classes in which co-occurrence of anti-patterns and clones in comparison with other classes in the analyzed software projects. S2 [7] also confirmed that classes with code smells tend to be more change- and fault-prone than other classes, and that this is even more evident when classes are affected by multiple smells [7]. The authors reported high fault-proneness for the code smell *Message Chain*. Of the 395 releases analyzed of 30 projects, the smell *Message Chain* affected 13%, and in the most affected release (a release of `HSQLDB`) only four out of the 427 classes (0.9%) are instances of this smell. Therefore, the authors concluded although the *Message Chain* smell is potentially harmful its diffusion is fairly limited. S3 emphasized that *Message Chain* increased the flaws in two software projects (`Eclipse` and `ArgoUML`). However, authors informed that when detected in larger files, the number of detected flaws were actually smaller.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at *Information* **2018**, *9*, 273; doi:10.3390/info9110273

S3 [8] reported the influence of five code smells (*Data Clumps*, *Message Chains*, *Middle Man*, *Speculative Generality* and *Switch Statements*) on the fault-prone behavior of source code. However, the effect that these smells have on faults seems to be small [8]. The authors argued that, in general, except for the the *Message Chains*, code smells analyzed have a relatively small influence (always below 10%) on bugs, suggesting that *Switch Statements* had no effect in any of the three analyzed software projects, as *Data Clumps* reduced the failures in `Apache` and `Eclipse` software projects, but increased failures in the `ArgoUML` software project. *Middle Man* reduced flaws only in `ArgoUML` and *Speculative Generality* reduced flaws only in `Eclipse`. In that study, collected evidence suggested that smells have different effects on different systems and that arbitrary refactoring is not guaranteed to reduce fault-proneness significantly. In some cases, it may even increase fault-proneness [8].

S4 [9] reported evidence of the influence of the code smells *Shotgun Surgery*, *God Class* and *God Method* on bugs. The authors analyzed the post-release system evolution process to reach this conclusion [9]. Results confirmed that some bad smells were positively associated with the class error probability in three error-severity levels (High, Medium and Low) usually applied to classify issues. This finding indicated the possibility to use code smells as a systematic method to identify and refactor problematic classes in this specific context [9]. The study reported that the code smells (*Shotgun Surgery*, *God Class* and *God Method*) were positively associated (ratio greater than one) with class error probability across and within the three error-severity levels (High, Medium and Low) created by the authors representing the original `Bugzilla` levels (`Blocker` and `Critical`, `Major`, `Normal` and `Minor`). In particular, *Shotgun Surgery* was associated with all severity levels of errors in all analyzed releases of the `Eclipse` software project. On the other hand, the study did not find relevant relationship between smells *Data Class* and *Refused Bequest* with the occurrence of bugs.

Although not explicitly investigating the occurrence of bugs as a consequence of code smells, the study S5 [24] analyzed the influence of a set of code smells in class change-proneness. The literature provided evidence based on a plethora of previous research arguing that change-proneness increase the probability of the inclusion of bugs in software projects [43–46]. Considering this scenario, we included study S5 in this SLR. Twenty nine code smells were analyzed in S5 [24] throughout 9 releases of the `Azureus` software project and in thirteen releases of the `Eclipse` software project to investigate to which extent these code smells influenced class change-proneness. The authors concluded that, in almost all releases of `Azureus` and `Eclipse` software projects, classes with code smells were more change-prone than others, and that specific smells were more correlated than others to change-proneness [24]. In the case of the `Azureus` software project, the smell *Not Abstract* had a significant impact on change proneness in more than 75% of releases, whereas *Abstract Class* and *Large Class* resulted to be significant in more than 50% of the analysed releases. In the `Eclipse` software project, the smells that had a significant effect on change-proneness for 75% of the releases or more were *HasChildren*, *MessageChainsClass*, and *NotComplex*.

The authors of S7 [11] provided evidence that instances of *God Class* and *Brain Class* were changed more frequently and contained more defects than classes not affected by those smells. However, they also argue that when normalized the measured effects with respect to size, *God Class* and *Brain Class* were less subject to change and had fewer defects than other classes [11]. Considering that both *God Class* and *Brain Class* have tendency to increase their size, there is also a tendency for more defects in these classes

S9 [26] reports that empirical evidence suggests that `JavaScript` files without code smells have hazard rates 65% lower than `JavaScript` files with code smells. The conclusion is that the survival of `JavaScript` files against the occurrence of faults increases with time if the files do not contain code smells [26].

For S11 [13], the results of the empirical study also indicated that classes affected by code smells are more likely to manifest bugs. The study analyzed the influence of the code smells *Schizophrenic Class* and *Tradition Breaker* on the occurrence of bugs. The first is associated to a considerable proportion of bugs, whereas *Tradition Breaker* seemed to have a low influence on the bugs detected. The authors

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at *Information* **2018**, *9*, 273; doi:10.3390/info9110273

recommend investigating these smells in other systems, with an emphasis on *Schizophrenic Class*. Still according to S11 [13], empirical evidence from the `Apache Ant` software project indicated that classes affected by code smells are up to three more likely to show bugs than other classes. In the case of the `Apache Xerces` software project, the odds rate is up to two times more likely to show bugs.

S13 [47] reported that one of the most predominant kind of performance-related change is the fixing of bugs as a consequence of code smells manifested in the code. The study provided examples of new releases of software project to fix inefficient usage of regular expressions, recurrent computations of constant data and usage of deprecated decryption algorithms [47].

In S14, authors reported that in 34 analyzed software projects there was a significant positive correlation between number of bugs and number of anti-patterns. The study reported a negative correlation between number of anti-patterns and maintainability. This further supports the intuitive thinking that consider anti-patterns, bugs and (lack of) quality as related.

Study S18 [12] investigated the association of code smells with merge conflicts, i.e., the impact on the bug proneness of the merged results. The authors of S18 argued that program elements that are involved in merge conflicts contain, on average, 3 times more code smells than program elements that are not involved in a merge conflict [12]. In S18, 12 out of the 16 smells that co-occurred with conflicts are significantly associated to merge conflicts. From those, *God Class*, *Message Chain*, *Internal Duplication*, *Distorted Hierarchy* and *Refused Parent Bequest* stood out. Still in S18, the sole (significant) smell associated with semantic conflicts are *Blob Operation* and *Internal Duplication* and are respectively 1.77 times and 1.55 times more likely to be present in a semantic conflict than with non-semantic conflicts.

As shown in Figure 3, the smell *God Class* stood out from the others as its influential role on bug occurrence in the projects analyzed in studies S4, S7, S11 and S18. In S4, *God Class* is positively correlated to code fault-proneness in three releases of the `Eclipse` project. Likewise, the code smell *Message Chain* has also distinguished as influential on bug occurrences as reported in studies S2, S3, S5, S11, S18. On the other hand, no relevant tendency to change- and fault-prone was reported as a consequence of the following code smells *Data Class*, *Refused Bequest*, *Not Informed*, *Tradition Braker*, *Data Clumps*, *Middle Man* and *Switch Statements*. As can be seen in the same figure, there is a node labeled as *Not Informed* connected to *Less Influential Code Smells*. In this case, studies S1, S2, S5, S8, S9, S13 and S14 did not informed which code smells were less influential in their respective studies.

S11 reported *God Class* as the smell with the greatest number of related bugs, with a percentage of 20%, followed by *Feature Envy* with a percentage close to 15%, *Schizophrenic Class* with 9%, *Message Chain* with 7% in the analyzed projects. Still in S11, *Tradition Breaker*, *Data Clumps* and *Data Class* were associated to percentages equal or lower than 5%. *Feature Envy* stood apart in S8. *Message Chains* also featured prominently in S2, S3, S5 and S18. In S3, *Message Chains* was associated to higher occurrence of faults in the software projects `Eclipse` and `ArgoUML`, though no influence was detected in the software project `Apache Commons`. *Middle Man* was related to fewer faults in `ArgoUML`, while *Data Clumps* was related to fewer faults in `Eclipse` and `Apache Commons`. Also according to S3, *Switch Statements* showed no effect on faults in any of the three systems. In S5, the smell *NotAbstract* was the sole smell to betray a significant impact on change proneness in more than 75% of the software project `Azureus`. *AbstractClass* and *LargeClass* proved to have a significant influence in more than 50% of the releases (5 out of 9) in project `Azureus`, according to S5. In `Eclipse`, the smells *HasChildren*, *MessageChainsClass* and *NotComplex* had a significant effect on change-proneness for more of 75% of the releases. In S9, the risk rate varied between the five software projects analised, since the smell *Chained Methods*, *This Assign* and *Variable Re-assign* had the highest hazard ratios in the `Express` software project; the smells *Nested Callbacks*, *Assignment in Conditional Statements* and *Variable Re-assign* had the highest hazard rates in the `Grunt` software project; the smells *Deeply Nested Code* was the most hazardous smell in terms of bug influence in the `Bower` software project; the smell *Assignment in Conditional Statements* had the highest hazard ratio in `Less.js` and the smell *Variable Re-assign* in `Request` software project in terms of bug influence.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at *Information* **2018**, *9*, 273; doi:10.3390/info9110273

Studies S6, S12, S16 (*Comments* code smell) and S17 (*Code Clone* code smell) focused on one specific smell. The authors concluded that they have an influence on bug occurrence in the analyzed projects. Therefore, they were considered as more influential. S1, S13, S14 did not disclosed which smells exerted the greatest influence on bug occurrence. S1, S2, S5, S8, S9, S13, S14 did not informed which code smells were less influential.

S6, S12 and S16 analyzed the *Code Comments* smell. The S12 results suggest a tendency for the presence of inner comments to relate to fault-proneness in methods. However, more inner comments do not seem to result in higher fault-proneness. For S6 and S16, comments are helpful in improving the comprehensibility of source code, but its motivation can also be to ameliorate the lack of comprehensibility of complicated and difficult-to-understand code sections. Consequently, some well-written comments may be indicative of low-quality code. For S6, methods having more comments than the quantity estimated by size and complexity are about 1.6-2.8 times more likely to be faulty than average. S16 also suggested that the risk of being faulty in well commented modules is about 2 to 8 times greater than in non commented modules.

Unlike S1, S2, S3, S4, S5, S6, S7, S8, S9, S11, S12, S13, S14, S16, S17, S18 the S10 study concluded that code smells do not increase the proportion of bugs in software projects. For S10, a total of 137 different problems were identified, of which only 64 problems (47%) were associated with the source code, with code smells representing only 30% of those 47% of problems. The other 73 problems (53%) were related to other factors, such as: lack of adequate technical infrastructure, developer coding habits, dependence on external services such as Web services, among others. S10 concluded that code smells are only partial indicators of maintenance difficulties, because the study results showed a relatively low coverage of smells when observing project maintenance in general. S10 commented that analyzing code smells individually can provide a wrong picture, due to potential interaction effects among code smells and between smells and other features such as the ones aforementioned. The authors of S10 suggested that to evaluate code more widely and safely, different analysis techniques should be combined with code smell detection.

S15 is in line with S10 and reported that: 1) most bugs had little to do with clones; 2) cloned code contained less "buggy code" (i.e., code implicated in bug fixes) than the rest of the system; 3) larger clone groups did not have more bugs than smaller clone groups, and in fact, making more copies of code does not introduce more defects; and furthermore, larger clone groups had lower bug density per line than smaller clone groups; 4) scattered clones across files or directories may not induce more defects; and 5) bugs with high clone content may require less effort to fix (as measured in number of lines changed to fix a bug) and most of the bugs (more than 80%) have no cloned code and around 90% of bugs have clone ratio lower than the average project. In other words, both S10 and S15 diverge from the other selected studies due to not finding evidence of clear relationship among code smells and software bugs.

5.2. Tools, Resources and Techniques to Identify the Influence of Specific Code Smells on Bugs

In this section, we use evidence collected from the selected studies to answer research question **RQ2**: *Which tools, resources and techniques have been adopted to identify evidence of the influence of code smells on the occurrence of software bugs?* Evidence collected from selected studies is presented in Figure 4.

Figure 4 depicts tools, resources and techniques reported in the selected studies to investigate the influence of specific code Smells on software bugs. S1 evaluated fault-proneness by analyzing the change report and fault report included in software repertories. S1 used the version-control systems `CVS`[1], also used by S5 and S8, and `Subversion`[2], also used in studies S7, S10, S11 and S17. S1 executed a `Perl` script that implemented the heuristics proposed in [48] to analyze the commit

---

[1] http://cvs.nongnu.org/
[2] http://subversion.apache.org/

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at Information 2018, 9, 273; doi:10.3390/info9110273

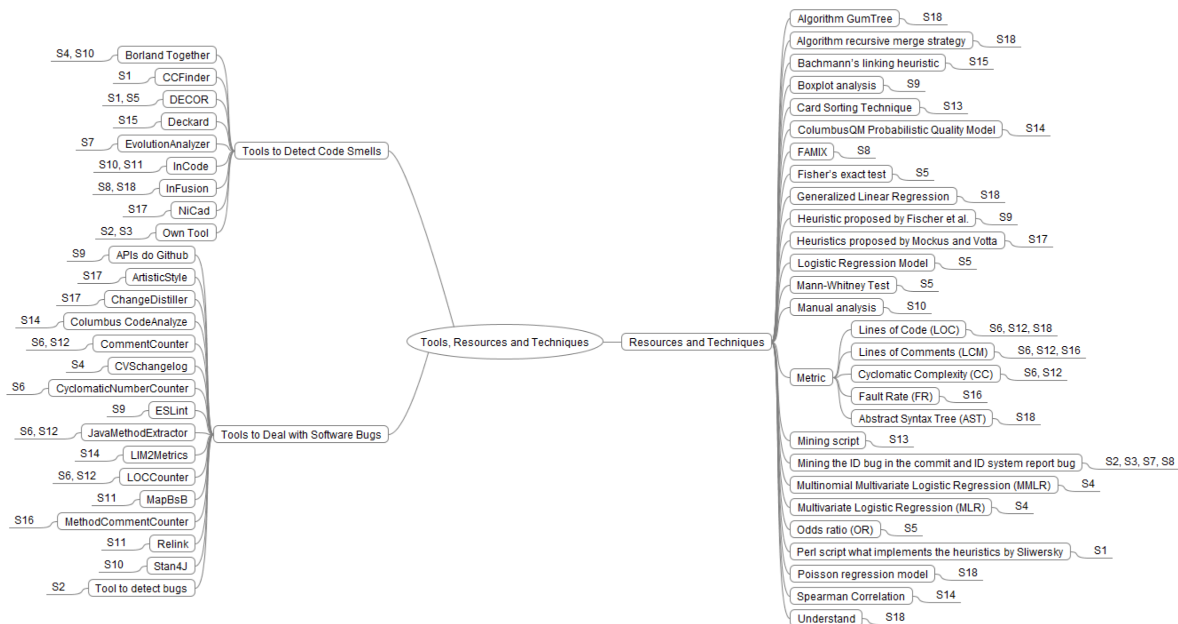

Figure 4. Tools, Resources and Techniques to Identify the Influence of Specific Code Smells on Bugs

message co-occurrence with maintenance activities to detect bug-fixing commits. Those heuristics searched for commit messages containing words such as "bug" and "fault" and performed during the maintenance phase of the studied release of a system. The bug identifier found in the commit log message is then compared with the projects list of known bugs to determine the list of files (and classes) that were changed to fix a bug. Finally, the script checked the excerpt of changes performed on this list of files using `CCFinder` and `DECOR` (Defect Detection for CORrection) tools to identify the files in which a code smell and a fault occurred in the same file location. S5 also used `DECOR` to detect smells and to test whether the proportion of classes exhibiting at least one change, significantly varied between classes with smells and other classes. For that purpose, they used Fisher's exact test [49], which checks whether a proportion varies between two samples and also compute the odds ratio (OR) [49] indicating the likelihood of an event to occur. To compare the number of smells in change-prone classes with the number of smells in non-change-prone classes, a (non-parametric) Mann-Whitney test was used and to relate change-proneness with the presence of particular kinds of smells, a logistic regression model [50] was used, similarly to Vokac's study [51].

In S2, S3, S4, S7, S8, S11, S15, the `Bugzilla` repositories were used to query the bugs of the projects under analysis, while S11, S15 and also S2 and S7 for some projects used the `Jira` tracking system. S15 used Git. S2, S3, S9, S14, S16, S4, do not report the version control system used. S2, S3 and S7 opted to develop their own detection tool and only S7 identifies the tool, naming it `EvolutionAnalyzer` (EvAn). S2 and S3 opted for the development tool as none of them were ever applied to detect all the studied code smells. For S2, their tool detection rules are generally more restrictive to ensure a good compromise between recall and precision, with the consequence that they may miss some smell instances. To validate this claim, they evaluated the behavior of three existing tools: `DECOR`, `JDeodorant` and `HIST`. S15 used `Deckard` [52]. From the `Deckard` output, extract filename, line number, which clone a line belongs to and the sibling clones. S8 parsed the source code with the tool `inFusion`[3] and produced a `FAMIX`[4]-compliant object-oriented model of the system whose result is a list of method-level design flaws from each class. Having the FAMIX-compliant model as input, the authors used detection

[3] http://www.intooitus.com/inFusion.html
[4] `FAMIX` is a language independent object-oriented meta-model of software system [53]

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at *Information* **2018**, *9*, 273; doi:10.3390/info9110273

strategies to spot the design flaws. This operation is performed within the Moose reengineering framework [54] and the result is a list of method level design flaws that each class contains. To identify a relationship between bugs and source code, S2, S3 and S8 resorted to a mining technique of regular expressions applied to bug fixing commits written by developers, which often include references to problem reports containing issue IDs in the versioning system change log, e.g., "fixed issue ID", "issue ID", "bug", etc. These can then be linked back to issue tracking system's issue identifier [55,56]. In S8, if looking for the mere number, it is possible to obtain false positives. Thus, in their algorithm, each time a candidate reference to a bug report is found, a check is made that a bug with such an id exists and also a verification that the date in which the bug was reported is prior to the timestamp of the commit comment in which the reference was found (i.e., it verifies that the bug is fixed after being reported).

S4 and S10 used `Borland Together` [5] - a plug-in tool for `Eclipse` - to identify the classes that had the bad smells. S10 also used `InCode`[6]. To analyse the change reports from the `Eclipse` change log, S4 used `CVSchangelog`, a tool available at `Sourceforge`. S4 used two types of dependent variables for their experiment: a binary variable indicating whether a class is erroneous or not and a categorical variable indicating the error-severity level. They used the Multivariate Logistic Regression (MLR) to study the association between bad smells and class error proneness and the Multinomial Multivariate Logistic Regression (MMLR) to study the association between bad smells and the various error-severity levels.

S10 used `Trac`[7], a system similar to `Bugzilla`. Observation notes and interview transcripts were used to identify and register the problems, and where applicable, the Java files associated to the problems were registered. The record of maintenance problems was examined and categorized into non-source code related and source code-related. Smells were detected via `Borland Together` and `InCode`[8]. Files that did not contain any detectable smell were manually reviewed to see if they exhibited any characteristics or issues that could explain why they were associated to problems. This step is similar to doing code reviews for software inspection, where peers or experts review code for constructs that are known to lead to problems. In this case, the task is easier because it is already known that there is a problem associated with the file. It is just a matter of looking for evidences that can explain the problem. In addition, to determine whether multiple files contributed to maintenance problems, the notion of *coupling* was used as part of the analysis. Tools `InCode` and `Stan4J`[9] were used to identify such couplings.

S6, S12 and S16 conducted an empirical analysis of relationships between comments and fault-proneness in the programs. S12 focused on comments describing sequences of executions, i.e., *Functions/Methods* and *Documentation Comments* and *Inner Comments*. S6 studied *Lines of Comments* (LCM) written inside a method's body and S16 studied *Lines of Comments* written in modules. S12 and S6 analysed projects maintained with `Git`, which provides various powerful functions for analysing their repositories. For example, the "`git log`" command can easily and quickly extract commit logs which specific keywords (e.g. "bug"). `Git` can also easily make a clone of the repository in one's local hard disk, so that data collection can be quickly performed with low cost. S16 did not disclose the control system used. The failures of the projects analysed in S12, are kept in `Git`. S6 did not disclose the failure portal. S16 got its fault data from the `PROMISE` data repository, also used by S16.

To analyze relationships between comments and fault-proneness, S6, S12 and S16 collected several metrics. For each method S12 calculated *Lines of Inner comments* (LOI) and *Lines of Documentation comments* (LOD) for capturing the amount of comments in a method. Then, they collected the change

---

5 http://www.borland.com/us/products/together
6 http://www.intooitus.com/products/incode
7 http://trac.edgewall.org
8 http://www.intooitus.com/products/incode
9 http://stan4j.com

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at Information 2018, 9, 273; doi:10.3390/info9110273

history of each source file from Git, and extracted the change history of each method in the file. They obtained the latest version of the source files and make the complete list of methods appeared in the source files, except for abstract methods. They used tool JavaMethodExtractor to extract the methods. For each method, they collected its change history. They examined whether the method was changed by checking all commits corresponding to the source file which the target method is declared in. Let a method be faulty if one of its changes was a bug fixing.

For data collection, S6 used tools JavaMethodExtractor, CommentCounter, LOCCounter, and CyclomaticNumberCounter and obtained the latest version of the source files from the repository, and make a list of the methods which appear in the files, except for all abstract methods. Then, get the "initial" version of the methods by tracing their histories of changes, and measure the LCM, LOC and CC values in those initial version of methods. Moreover, for each method, examine whether the method is faulty or not by checking all changes in which the method was involved. S6 and S12 decided a change to be a bug fixing when the corresponding commit's log includes one of bug-fix-related words—"bug, fix, defect." while S16 uses metric the *Lines of Comments* (LCM) for computing the amount of comments written in a module, and metric FR [57] for the fault-proneness of modules.

S9 implemented all the steps of approach into a framework available on GitHub[10] to detect code smells in JavaScript. All the five studied systems are hosted on GitHub and use it as their issue tracker. The framework performs a Git clone to get a copy of a system's repository and then generates the list of all the commits for used to perform analysis at commits level. S9 used GitHub APIs to get the list of all the resolved issues on the systems. They leveraged the SZZ algorithm [48] to detect changes that introduced faults. Identify fault-fixing commits using the heuristic proposed by Fischer et al. [55], which consists in using regular expressions to detect bug IDs from the studied commit messages. Next, they extracted the modified files of each fault-fixing commit through the Git command. Given each file F in a commit C, they extract C's parent commit C0. Then, they used Git's diff command to extract F's deleted lines. They applied Git's blame command to identify commits that introduced these deleted lines, noted as the "candidate faulty changes". Finally, they filtered the commits that were submitted after their corresponding bugs' creation date. To automatically detect code smells in the source code, the *Abstract Syntax Trees* (AST) was first extracted from the code, using ESLint[11], a popular open source lint utility for JavaScript as the core of the framework. They developed their own plugins and modified ESLint built-in plugins to traverse the AST generated by ESLint to extract and store information related to the set of code smells. For each kind of smell, a given metric was used. To identify code smells using the metric values provided by the framework, they defined threshold values above which files should be considered as having the code smell using *Boxplot* analysis. To assess the impact of code smells on the fault-proneness of JavaScript files, they performed survival analysis, comparing the time until a fault occurrence, in files containing code smells and files without code smells. Survival analysis was used to model the time until the occurrence of a well-defined event [58].

The goal of S13 is to investigate performance-related commits in Android apps, with the purpose of understanding their nature and their relationship with projects characteristics, such as project domain or size. The study context consists of 2,443 open-source apps the Google Play store and their evolution history, have their versioning history hosted on GitHub. S13 extracted the commits and pCommits (the number of performance-related commits in the GitHub repository of the app, as compared to the overall number of commits, and (ii) the app category on Google Play) using a script that considers only the folder containing the source code and resources of the mobile app, excluding back end, documentation, test or mockups. The mining script identifies a commit as performance-related if it matches at least one of the following keywords: wait, slow, fast, lag, tim,

[10] https://GitHub.com/amir-s/smelljs
[11] http://eslint.org/

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at Information 2018, 9, 273; doi:10.3390/info9110273

minor, stuck, instant, respons, react, speed, latenc, perform, throughput, hang, memory, leak. These keywords have been identified by considering, analysing, and combining mining strategies in previous empirical studies on software performance in the literature. They extracted the category variable by mining the web page of the Google Play store of each app. Then, they identified the concerns by applying the open card sorting technique [59] to categorize performance-related commits into relevant groups. They performed card sorting in two phases: in the first phase, they tagged each commit with its representative keywords (e.g., read from file system, swipe lag) and in the second phase they grouped commits into meaningful groups with a descriptive title (e.g., UI issues, file system issues).

For the purpose of quality assessment, S14 chose the ColumbusQM probabilistic quality model [60], which ultimately produces one number per system describing how "good" that system is. The antipattern-related information came from their own, structural analysis based extractor tool and source code metrics were computed using Columbus CodeAnalyzer reverse engineering tool [61]. S14 compiled the types of data described above for a total of 228 open-source Java systems, 34 of which have corresponding class level bug numbers from the open- access PROMISE database. The metric values were extracted by Columbus. First, they converted the code to the LIM model (Language Independent Model), a part of the Columbus framework. From this data, tool LIM2Metrics can compute various kinds of source code metrics. They performed correlation analysis on the collected data. Since they did not expect the relationship between the inspected values to be linear - only monotone - they used Spearman correlation, which is in fact a "traditional" Pearson correlation. The extent of this matching movement is somewhat masked by the ranking - which can be viewed as a kind of data loss - but this is not too important as they are more interested in the existence of this relation rather than its type.

S11 obtained the SVN change log and Bugzilla bug report of the selected projects. The source code for each version of the software was also extracted and processed by InCode[12] for bad smells detection. To help with information processing, tool MapBsB was developed, whose name is an acronym for *Bad Smells Mapping for Bugs*, which generates a set of scripts to support the mapping of bad smells and bugs to classes. These scripts are processed by the Relink[13] tool, which in turn generates a file with bugs and revisions (commits) associated. Finally, bad smells, bugs-revisions and change log files are processed so that one can analyse software versions, link bugs to system classes and versions, and cross relate information between from bad smells and bugs.

S17 extracted the SVN commit messages by applying SVN log command, to identify bug-fix commits of a candidate systems and apply the heuristics proposed by Mockus and Votta [62] on the commit messages to automatically identify bug-fix commits. They analysed and identified all the cloned methods that are related to bug-fixes and analysed the stability considering fine-grained change types associated with each of the bug-related clone fragments to measure the extent of the relationship between stability and bug-proneness. Pretty-printing of the source files was then carried out to eliminate the formatting differences using tool ArtisticStyle [14] and extract the file modification history using SVN diff command to list added, modified and deleted files in successive revisions. To analyse the changes to cloned methods throughout all the revisions, they extract method information from the successive revisions of the source code. They store the method information (file path, package name, class name, signature, start line, end line) in a database to use for mapping changes to the corresponding methods.

S18 selected active open source projects hosted on GitHub, which use the Maven build system and were developed in Java. They chose to use InFusion to identify code smells. Since Git does not record information about merge conflicts, they recreated each merge in the corpus in order to determine if a conflict had occurred. They used Git's default algorithm, the recursive merge strategy, as this is the

---

[12] Intooitus, the company that provided InCode, seems to have closed and the tool website is no longer available.
[13] Available at https://code.google.com/archive/p/bugcenter/wikis/ReLink.wiki
[14] https://sourceforge.net/projects/astyle/

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at Information 2018, 9, 273; doi:10.3390/info9110273

most likely to used by the average Git project. They use GumTree for their analysis, as it allows us to track elements at AST level. This way they could track just the elements that we are interested in (statements) and ignore other changes that do not actually change the code. To identify the impact of code smell on the number of bug-fixes that occur on lines of code that are associated with a code smell and a merge conflict. We use Generalized Linear Regression. The dependent variable (count of bug fixes occurring on smelly and conflicting lines) follows a Poisson distribution. Therefore, we use a Poisson regression model with a log linking function. We use Understand to count the number of references to, and from other files to the files that are involved in a conflict. They also collected the following factors for each commit such as the difference between the two merged branches in terms of LoC, AST difference, and the number of methods and classes being affected. After collecting these metrics, they checked for multi-collinearity using the Variance Inflation Factor (VIF) of each predictor in our model. VIF describes the level of multicollinearity (correlation between predictors). A VIF score between 1 and 5 indicates moderate correlation with other factors, so they selected the predictors with VIF score threshold of 5.

5.3. Software Projects Analysed in the Selected Studies

Table 4 presents the software projects analyzed in each study that discussed the relationship between code smells and software bugs. The vast majority of the analyzed software projects were developed in the Java language, except the ones from studies S15 (C language) and S9 (JavaScript). Moreover, all the analysed projects are open source, except the one used in study S10 classified as proprietary.

Project	Classification	Language	Study
Eclipse	Open source	Java	S1, S3, S4, S5, S16
Apache Xerces	Open source	Java	S2, S7, S11
Apache Ant	Open source	Java	S2, S11, S16
Apache Lucene	Open source	Java	S2, S7, S8
Apache Tomcat	Open source	Java	S11, S16
ArgoUML	Open source	Java	S2, S3
Azureus	Open source	Java	S1, S5
Checkstyle Plug-in; PMD; SQuirreL SQL Client; Hibernate ORM	Open source	Java	S12, S6
JHotDraw	Open source	Java	S1, S2
Apache Commons	Open source	Java	S3
Apache Jmeter; Apache Lenya	Open source	Java	S11
Apache httpd; Nautilus; Evolution; GIMP	Open source	C	S15
Software Project A; Software Project B; Software Project C; Software Project D	Proprietary	Java	S10
Request; Less.js; Bower; Express; Grunt	Open source	JavaScript	S9
Log4J	Open source	Java	S7
Maven; Mina; Eclipse CDT; Eclipse PDE UI; Equinox	Open source	Java	S8
aTunes; Cassandra; Eclipse Core; Elastic Search; FreeMind; Hadoop; Hbase; Hibernate; Hive; HSQLDB; Incubating; Ivy; JBoss; Jedit; JFreeChart; jSL; JVlt; Karaf; Nutch; Pig; Qpid; Sax; Struts; Wicket; Derby	Open source	Java	S2
DNSJava; JabRef; Carol; Ant-Contrib; OpenYMSG	Open source	Java	S17
2443 apps (undisclosed names)	Open source	Not informed	S13
34 projetos (undisclosed names)	Open source	Java	S14
143 projetos (undisclosed names)	Open source	Java	S18

Table 4. Software Projects Analysed in the Selected Studies

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at Information 2018, 9, 273; doi:10.3390/info9110273

6. Threats to Validity

The following types of validity issues were considered when interpreting the results from this review.

Conclusion validity. There is a risk of bias in the data extraction. This was addressed by defining a data extraction form to ensure consistent extraction of relevant data to answering the research questions. The findings and implications are based on the extracted data.

Internal validity. One possible threat is selection bias. We addressed this threat during the selection step of the review, i.e., the studies included in this review were identified by means of a thorough selection process comprising multiple stages.

Construct validity. The studies identified in the SLR were accumulated from multiple literature databases covering relevant journals and proceedings. One possible threat is bias in the selection of publications. This is addressed by specifying a research protocol that defines the research questions and objectives of the study, inclusion and exclusion criteria, search strings that we intend to use, the search strategy and a strategy for data extraction. Another potential bias is related to the fact that code smells were identified through the use of specific detection tools. This has the possibility of not drawing a complete picture because not all code smells could have been detected and hence their influence on software bugs. The goal is not to obtain an exhaustive list of code smells that influence the occurrence of software bugs, but at least the ones that are more representative in this influence and hence in the degradation of the quality of the analyzed software qualities.

7. Conclusion

As far as we know, this is the first systematic review of literature (SLR) that addresses the issue of the influence of code smells on software project bugs. The objective of this study was to identify evidence in the literature about how code smells influence the occurrence of bugs in software projects, as well as to identify the tools, resources and techniques used in the studies. We selected 18 studies published in three digital libraries. These studies presented analyzes of open source and proprietary projects, as well as various types of code smells and anti-patterns.

With the analysis of the results of the selected studies, it was possible to answer the two research questions of this SLR. For **RQ1**, of the 18 studies, 16 presented evidence that code smells influenced the occurrence of project bugs, as well as some studies highlighted code smells that had a greater or lesser influence on the bugs and only 2 concluded that the code smells did not influence bugs. For **RQ2**, we identified tools that detect code smells, and complementary tools that can be used in the analyzes, as well as techniques and resources that were adopted in the stages of the studies, and ways of relating the source code of software projects with the bugs registered in the portals.

The evidence gathered in the results of the selected studies showed that code smells exert an influence on bugs, that is, classes affected by code smells are more prone to changes and contain failures than classes that have not been affected, as well as, there are tools, resources and techniques that may have been used.

With the analysis of the selected studies, a limitation was identified in almost all the presented results. The studies did not present the characteristics of the bugs that were related to the classes affected by code smells. As a future work, we aim to analyze the influence of code smells on bugs, in open source projects developed in Java language and that the bug portal is Bugzilla, and present the characteristics of severities and priorities of related bugs the classes affected by code smells.

Author Contributions: Aloisio Sampaio Cairo and Glauco de Figueiredo Carneiro together searched for eligible papers from the publication databases and read the eligible papers carefully. Aloisio Sampaio Cairo, Glauco de Figueiredo Carneiro and Miguel P. Monteiro wrote the article. All authors have read and approved the final manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at *Information* **2018**, *9*, 273; doi:10.3390/info9110273

Appendix A

The list of the selected papers of this Systematic Literature Review is provided in the following Tables A1 and A2.

ID	Author, Title	Venue	Year
S1 [5]	Jaafar, F.; Lozano, A.; Guéhéneuc, Y.-G.; Mens, K. *On the analysis of co-occurrence of anti-patterns and clones*	QRS	2017
S2 [7]	Palomba, F.; Bavota, G.; Di Penta, M.; Fasano, F.; Oliveto, R.; Lucia, A.D. *On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation*	ESENF	2017
S3 [8]	Hall, T.; Zhang, M.; Bowes, D.; Sun, Y. *Some code smells have a significant but small effect on faults*	ATSME	2014
S4 [9]	Li, W.; Shatnawi, R. *An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution*	JSSOD	2007
S11 [13]	Nascimento, R.; Sant'Anna, C. *Investigating the Relationship Between Bad Smells and Bugs in Software Systems*	SBCARS	2017
S12 [57]	Aman, H.; Amasaki, S.; Sasaki, T.; Kawahara, M. *Empirical Analysis of Fault-Proneness in Methods by Focusing on their Comment Lines*	APSEC	2014
S13 [47]	Das, T.; Di Penta, M.; Malavolta, I. *A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps*	ICSME	2016
S14 [63]	Bán, D.; Ferenc, R. *Recognizing antipatterns and analyzing their effects on software maintainability*	ICCSA	2014
S15 [1]	Rahman, F.; Bird, C.; Devanbu, P. *Clones: What is that smell?*	MSR	2012
S16 [64]	Aman, H. *An Empirical Analysis on Fault-proneness of Well-Commented Modules*	IWESEP	2012
S17 [4]	Rahman, M.S.; Roy, C.K. *On the Relationships between Stability and Bug-Proneness of Code Clones: An Empirical Study*	SCAM	2017

Table A1. List of selected studies from the Search String

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at *Information* **2018**, *9*, 273; doi:10.3390/info9110273

ID	Author, Title	Forward/backward snowballing	Venue	Year
S5 [24]	Khomh, F.; Di Penta, M.; Guéhéneuc, Y.-G. *An Exploratory Study of the Impact of Code Smells on Software Change-proneness*	Backward of S2	WCRE	2009
S6 [65]	Aman, H.; Amasaki, S.; Sasaki, T.; Kawahara, M. *Lines of Comments as a Noteworthy Metric for Analyzing Fault-Proneness in Methods*	Forward of S16	IEICE	2015
S7 [11]	Olbrich, S.M.; Cruzes, D.S. *Are all Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of three Open Source Systems*	Backward of S2	ICSM	2010
S8 [3]	D'Ambros, M.; Bacchelli, A.; Lanza M. *On the Impact of Design Flaws on Software Defects*	Backward of S2	QSIC	2010
S9 [26]	Saboury, A.; Musavi, P.; Khomh, F.; Antoniol, G. *An Empirical Study of Code Smells in `JavaScript` projects*	Forward of S2	SANER	2017
S10 [10]	Yamashita, A.; Moonen, L. *To what extent can maintenance problems be predicted by code smell detection? –An empirical study*	Backward of S3	Information and Software Technology	2013
S18 [12]	Ahmed, I; Brindescu, C; Ayda, U; Jensen, C; Sarma, A *An empirical examination of the relationship between code smells and merge conflicts*	Forward of S3	ESEM	2017

Table A2. List of selected studies from Snowballing

References

1. Rahman, F.; Bird, C.; Devanbu, P. Clones: What is that smell? *Empirical Software Engineering* **2012**, *17*, 503–530.
2. Fowler, M.; Beck, K. *Refactoring: improving the design of existing code*; Addison-Wesley Professional, 1999.
3. D'Ambros, M.; Bacchelli, A.; Lanza, M. On the impact of design flaws on software defects. Quality Software (QSIC), 2010 10th International Conference on. IEEE, 2010, pp. 23–31.
4. Rahman, M.S.; Roy, C.K. On the relationships between stability and bug-proneness of code clones: An empirical study. Source Code Analysis and Manipulation (SCAM), 2017 IEEE 17th International Working Conference on. IEEE, 2017, pp. 131–140.
5. Jaafar, F.; Lozano, A.; Guéhéneuc, Y.G.; Mens, K. On the Analysis of Co-Occurrence of Anti-Patterns and Clones. Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on. IEEE, 2017, pp. 274–284.
6. Khomh, F.; Di Penta, M.; Gueheneuc, Y.G. An exploratory study of the impact of code smells on software change-proneness. Reverse Engineering, 2009. WCRE'09. 16th Working Conference on. IEEE, 2009, pp. 75–84.
7. Palomba, F.; Bavota, G.; Di Penta, M.; Fasano, F.; Oliveto, R.; De Lucia, A. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* **2017**, pp. 1–34.
8. Hall, T.; Zhang, M.; Bowes, D.; Sun, Y. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **2014**, *23*, 33.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at Information 2018, 9, 273; doi:10.3390/info9110273

9. Li, W.; Shatnawi, R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software* **2007**, *80*, 1120–1128.
10. Yamashita, A.; Moonen, L. To what extent can maintenance problems be predicted by code smell detection?–An empirical study. *Information and Software Technology* **2013**, *55*, 2223–2242.
11. Olbrich, S.M.; Cruzes, D.S.; Sjøberg, D.I. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. Software Maintenance (ICSM), 2010 IEEE International Conference on. IEEE, 2010, pp. 1–10.
12. Ahmed, I.; Brindescu, C.; Mannan, U.A.; Jensen, C.; Sarma, A. An empirical examination of the relationship between code smells and merge conflicts. Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. IEEE Press, 2017, pp. 58–67.
13. Nascimento, R.; Sant'Anna, C. Investigating the relationship between bad smells and bugs in software systems. Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse. ACM, 2017, p. 4.
14. Rasool, G.; Arshad, Z. A review of code smell mining techniques. *Journal of Software: Evolution and Process* **2015**, *27*, 867–895.
15. Fernandes, E.; Oliveira, J.; Vale, G.; Paiva, T.; Figueiredo, E. A review-based comparative study of bad smell detection tools. Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering. ACM, 2016, p. 18.
16. Fontana, F.A.; Braione, P.; Zanoni, M. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology* **2012**, *11*, 5–1.
17. Van Emden, E.; Moonen, L. Java quality assurance by detecting code smells. Reverse Engineering, 2002. Proceedings. Ninth Working Conference on. IEEE, 2002, pp. 97–106.
18. Moha, N.; Gueheneuc, Y.G.; Duchien, L.; Le Meur, A.F. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering* **2010**, *36*, 20–36.
19. Marinescu, R. Detecting design flaws via metrics in object-oriented systems. Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on. IEEE, 2001, pp. 173–182.
20. Marinescu, R. Measurement and quality in object-oriented design. Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on. IEEE, 2005, pp. 701–704.
21. Marinescu, R. Detection strategies: Metrics-based rules for detecting design flaws. Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on. IEEE, 2004, pp. 350–359.
22. Lanza, M.; 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.
23. Mantyla, M.; Vanhanen, J.; Lassenius, C. A taxonomy and an initial empirical study of bad smells in code. Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on. IEEE, 2003, pp. 381–384.
24. Khomh, F.; Di Penta, M.; Guéhéneuc, Y.G.; Antoniol, G. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* **2012**, *17*, 243–275.
25. Visser, J.; Rigal, S.; Wijnholds, G.; van Eck, P.; van der Leek, R. *Building Maintainable Software, C# Edition: Ten Guidelines for Future-Proof Code*; " O'Reilly Media, Inc.", 2016.
26. Saboury, A.; Musavi, P.; Khomh, F.; Antoniol, G. An empirical study of code smells in javascript projects. Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on. IEEE, 2017, pp. 294–305.
27. Yamashita, A.; Moonen, L. Do developers care about code smells? an exploratory survey. Reverse Engineering (WCRE), 2013 20th Working Conference on. IEEE, 2013, pp. 242–251.
28. Lehman, M.M. Laws of software evolution revisited. European Workshop on Software Process Technology. Springer, 1996, pp. 108–124.
29. Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A.; Poshyvanyk, D. When and why your code starts to smell bad. Proceedings of the 37th International Conference on Software Engineering-Volume 1. IEEE Press, 2015, pp. 403–414.
30. Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; Poshyvanyk, D.; De Lucia, A. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering* **2015**, *41*, 462–489.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at *Information* **2018**, *9*, 273; doi:10.3390/info9110273

31. Tsantalis, N.; Chatzigeorgiou, A. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* **2009**, *35*, 347–367.
32. Sahin, D.; Kessentini, M.; Bechikh, S.; Deb, K. Code-smell detection as a bilevel problem. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **2014**, *24*, 6.
33. Khomh, F.; Vaucher, S.; Guéhéneuc, Y.G.; Sahraoui, H. A bayesian approach for the detection of code and design smells. Quality Software, 2009. QSIC'09. 9th International Conference on. IEEE, 2009, pp. 305–314.
34. Kessentini, W.; Kessentini, M.; Sahraoui, H.; Bechikh, S.; Ouni, A. A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Transactions on Software Engineering* **2014**, *40*, 841–861.
35. Danphitsanuphan, P.; Suwantada, T. Code smell detecting tool and code smell-structure bug relationship. Engineering and Technology (S-CET), 2012 Spring Congress on. IEEE, 2012, pp. 1–5.
36. Ambler, S.; Nalbone, J.; Vizdos, M. *Enterprise unified process, the: extending the rational unified process*; Prentice Hall Press, 2005.
37. Olbrich, S.; Cruzes, D.S.; Basili, V.; Zazworka, N. The evolution and impact of code smells: A case study of two open source systems. Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on. IEEE, 2009, pp. 390–400.
38. Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; De Lucia, A.; Poshyvanyk, D. Detecting bad smells in source code using change history information. Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, 2013, pp. 268–278.
39. BA, K.; Charters, S. Guidelines for performing Systematic Literature Reviews in Software Engineering **2007**. *2*.
40. Zhang, H.; Babar, M.A.; Tell, P. Identifying relevant studies in software engineering. *Information and Software Technology* **2011**, *53*, 625–637.
41. Dybå, T.; Dingsøyr, T. Empirical studies of agile software development: A systematic review. *Information and software technology* **2008**, *50*, 833–859.
42. Wohlin, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. Proceedings of the 18th international conference on evaluation and assessment in software engineering. ACM, 2014, p. 38.
43. Cotroneo, D.; Pietrantuono, R.; Russo, S.; Trivedi, K. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. *Journal of Systems and Software* **2016**, *113*, 27–43.
44. Qin, F.; Zheng, Z.; Li, X.; Qiao, Y.; Trivedi, K.S. An empirical investigation of fault triggers in android operating system. Dependable Computing (PRDC), 2017 IEEE 22nd Pacific Rim International Symposium on. IEEE, 2017, pp. 135–144.
45. González-Barahona, J.M.; Robles, G. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering* **2012**, *17*, 75–89.
46. Panjer, L.D. Predicting eclipse bug lifetimes. Proceedings of the Fourth International Workshop on mining software repositories. IEEE Computer Society, 2007, p. 29.
47. Das, T.; Di Penta, M.; Malavolta, I. A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps. Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on. IEEE, 2016, pp. 443–447.
48. Śliwerski, J.; Zimmermann, T.; Zeller, A. When do changes induce fixes? ACM sigsoft software engineering notes. ACM, 2005, Vol. 30, pp. 1–5.
49. Sheskin, D.J. *Handbook of parametric and nonparametric statistical procedures*; crc Press, 2003.
50. Hosmer, D.; Lemeshow, S. Applied logistic regression, John Wiley&Sons New York. *Applied logistic regression. 2nd ed. John Wiley & Sons, New York.* **2000**.
51. Vokac, M. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering* **2004**, *30*, 904–917.
52. Jiang, L.; Misherghi, G.; Su, Z.; Glondu, S. Deckard: Scalable and accurate tree-based detection of code clones. Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007, pp. 96–105.
53. Demeyer, S.; Tichelaar, S.; Ducasse, S. FAMIX 2.1—the FAMOOS information exchange model, 2001.
54. Nierstrasz, O.; Ducasse, S.; Gîrba, T. The story of Moose: an agile reengineering environment. *ACM SIGSOFT Software Engineering Notes* **2005**, *30*, 1–10.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 October 2018 doi:10.20944/preprints201810.0059.v1

Peer-reviewed version available at Information 2018, 9, 273; doi:10.3390/info9110273

55. Fischer, M.; Pinzger, M.; Gall, H. Populating a release history database from version control and bug tracking systems. Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on. IEEE, 2003, pp. 23–32.
56. Čubranić, D.; Murphy, G.C. Hipikat: Recommending pertinent software development artifacts. Proceedings of the 25th international Conference on Software Engineering. IEEE Computer Society, 2003, pp. 408–418.
57. Aman, H. Quantitative analysis of relationships among comment description, comment out and fault-proneness in open source software,”. *IPSJ Journal* **2012**, *53*, 612–621.
58. Fox, J.; Weisberg, S. *An R companion to applied regression*; Sage Publications, 2010.
59. Spencer, D. *Card sorting: Designing usable categories*; Rosenfeld Media, 2009.
60. Bakota, T.; Hegedűs, P.; Körtvélyesi, P.; Ferenc, R.; Gyimóthy, T. A probabilistic software quality model. Software Maintenance (ICSM), 2011 27th IEEE International Conference on. IEEE, 2011, pp. 243–252.
61. Ferenc, R.; Beszédes, Á.; Tarkiainen, M.; Gyimóthy, T. Columbus-reverse engineering tool and schema for C++. Software Maintenance, 2002. Proceedings. International Conference on. IEEE, 2002, pp. 172–181.
62. Mockus, A.; Votta, L.G. Identifying Reasons for Software Changes using Historic Databases. icsm, 2000, pp. 120–130.
63. Bán, D.; Ferenc, R. Recognizing antipatterns and analyzing their effects on software maintainability. International Conference on Computational Science and Its Applications. Springer, 2014, pp. 337–352.
64. Chen, J.C.; Huang, S.J. An empirical analysis of the impact of software development problem factors on software maintainability. *Journal of Systems and Software* **2009**, *82*, 981–992.
65. Aman, H.; Amasaki, S.; Sasaki, T.; Kawahara, M. Lines of comments as a noteworthy metric for analyzing fault-proneness in methods. *IEICE Transactions on Information and Systems* **2015**, *98*, 2218–2228.

Sample Availability: Samples of the compounds ...... are available from the authors.