information

Review

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

Aloisio S. Cairo [1], Glauco de F. Carneiro [1,*] and Miguel P. 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

Received: 1 October 2018; Accepted: 2 November 2018; Published: 6 November 2018

Abstract: Context: Code smells are associated to poor design and programming style, which often degrades code quality and hampers code comprehensibility and maintainability. Goal: identify published studies 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 selected studies from July 2007 to September 2017, which analyzed the source code of open source software projects and several code smells. Based on evidence of 16 studies covered in this SLR, we conclude that 24 code smells are more influential in the occurrence of bugs relative to the remaining smells analyzed. 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 that should be applied to analyze the influence of the smells. Conclusions: 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

Ever increasing maintenance costs are often a consequence of poor software design, bad coding style, and undisciplined practices. Among the consequences of such a scenario, error proneness stands out [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 and maintenance and promoting a negative impact on software quality and ease of evolution [3].

Code smells have been largely discussed by both the software engineering community and practitioners from the industry. There are several tools that support the detection of code smells in programs written in different languages [1,3–13]. Similarly, there are plenty of tools that deal with bug issue tracking. However, to the best of our knowledge, there are very few solutions that integrate data related to code smells and software bugs which allow a deeper analysis of how the first influences the occurrence of the second and vice versa. 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, cooperative-based approaches [14–16]. 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 a context of intensive use, this approach can be attractive for using automatic detection. Though many studies discussed code smells, just a relatively small number focused on the analysis of the influence of code smells on the occurrence of bugs 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). From the 18 selected studies, 16 studies reported the influence of code smells on the occurrence of bugs. Based on the analysis of these 16 studies, we identified 24 code smells with greater influence on bugs, specially *God Class*, *Comments*, *Message Chains*, and *Feature Envy* code smells. It was also possible to identify tools, techniques, and resources that were used by several studies. We identified the use of 25 tools, where 9 were focused on the detection of code smells, and the other 16 were focused on some stage of the empirical study reported in the study. We also identified the adoption of 24 techniques and resources such as metrics, heuristics, and scripts to perform the analysis presented in the selected studies.

The rest of this article is organized as follows. Section 2 presents the background related to code smells and software bugs. Section 3 outlines the research methodology. Section 4 discusses the results of the SLR and characteristics of the selected works, while the threats that could affect the validity of the presented results are discussed in Section 5. Section 6 concludes and presents plans for future work.

2. Code Smells and Software Bugs

According to Lehman's second law [17], as projects grow, they tend to increase in complexity. This growth favours the occurrence of new code smells in the evolving code. Fowler and Beck defined *code smells* as symptoms in the source code indicating that there may be deficiencies in the design and/or programming style of a software system [2]. They proposed a catalog of 22 code smells to be used as indicators of poor design and implementation choices [2]. As a catalog, these code smells can also serve as a vehicle to express what should be avoided to obtain good style and modularity in object-oriented programming. Code smells usually appear during change activities and when software bugs are fixed [18]. To tackle this problem, approaches and tools were implemented to support the identification of code smells [14–16,19–26].

Code smells are closely associated with the concept of refactoring, defined as a behaviour-preserving transformation in a program's source code [2]. Though programmers have been performing ad hoc transformations of their program's source code for decades, it became the subject of a formal study only in the 1990s. The earliest works on refactoring were carried out by Griswold and Notkin [27,28], focusing on block-structured imperative programs and functional programs. Opdyke and Johnson [29,30] were the first to use and coin the term *refactoring*, using it in the context of object-oriented frameworks. A few years later, the term *refactoring* became mainstream, specially as a result of the well-known book by Martin Fowler on the subject [2]. Fowler's book characterizes the term *refactoring* both as a noun and as a verb. As a noun, refactoring is "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [2]. As a verb, refactoring "restructures software by applying a series of refactorings without changing its observable behavior" [2]. The purpose of refactoring is to eliminate from the code negative symptoms characterized as code smells. Thus, for each code smell, there is a corresponding group of refactorings to remove it from the smelly code.

Depending on the situation, a code smell may concern a single method, a class, a group of classes, or an entire subsystem. For instance, Fowler's *Long Method* smell concerns a single method, characterizing situations in which the method is excessively long, making it difficult understand or to (re)use it, or usually both. Often, several different subtasks can be identified amid the method's code, which can be placed in their own methods by using the refactoring *Extract Method* multiple times. The code smell *Feature Envy* concerns at least two classes. It characterizes cases in which a given class seems more interested in a given class than the one it actually is in. Often, the way to remove this smell is a refactoring process comprised of multiple uses of the *Move Field* and *Move Method*, through which a better distribution of responsibilities between classes is attained. According to

Ambler et al. [31], agile methods adopted refactoring techniques to eliminate code smells and this way, reduce development and maintenance-related costs.

The catalog of smells initially proposed by Fowler [2] was subsequently extended with new code smells as reported in the following references [20,24,32–35]. This situation can be illustrated by the *God Class* and *Brain Class* smells proposed in [35] and derived from Fowler's original *Large Class* smell [2]. During the analysis of the selected studies to answer the research questions, we found out that some authors use different names for the code smells, other than those coined by Fowler in his catalog. In Table 1, we present the original name from Fowler's catalog along with the respective corresponding alternative names. Mantyla and colleagues [36] proposed a taxonomy to categorize similar bad smells. They argued that the taxonomy could help explain potential correlations between the subjective evaluations of the existence of the smells.

Many authors associate the occurrence of smells with poor quality which hampers software reusability and maintainability [5,9,10,13,14,35–39]. Software systems that have these symptoms are prone to the occurrence of bugs over time, increasing risks in the maintenance activities and therefore, contributing to a troublesome and costly maintenance process. Such situations often contribute to higher fault rates and more bugs filed by users and developers. Unfortunately, some developers still seem unaware of the potential harm caused by code smells [40].

Table 1. List of Code Smells according to Fowler [2] and 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)*

Code smells have been the target by researchers as a relevant theme in software engineering [24,41,42]. According to [5], classes betraying code smells are more likely to contain bugs compared to classes free of them [5]. Considering the relevance of this theme, 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.

3. Research Methodology

This section describes the methodology applied in the planning, conducting, and reporting phases of the review. In contrast to a non-structured review process, a Systematic Literature Review (SLR) [43], 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 following the steps conveyed in Figure 1.

Identify the needs for an SLR. Search for evidence in the literature of how code smells contribute to the occurrence of software bugs and which tools and/or resources are used by researchers and practitioners 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 what 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 in project releases and can be used to help them in mitigating and/or fixing them. *Research Question 2 (RQ2)*: *Which tools, resources, and techniques were used to find evidence of the influence of code smells on the occurrence of software bugs?* Knowledge of tools and resources used to identify evidence of this influence is an opportunity to motivate practitioners and researchers to use them in their projects.

3.1. 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. Though the literature includes publications related to code smell prior to 2007 (including the code smells catalogue [2]), we decided to focus on a time span of 10 years to identify relevant papers that have addressed and investigated potential influences of code smells on software bugs in the latest decade. The search string below was applied to three digital libraries, meeting the guidelines described in [43] 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").*

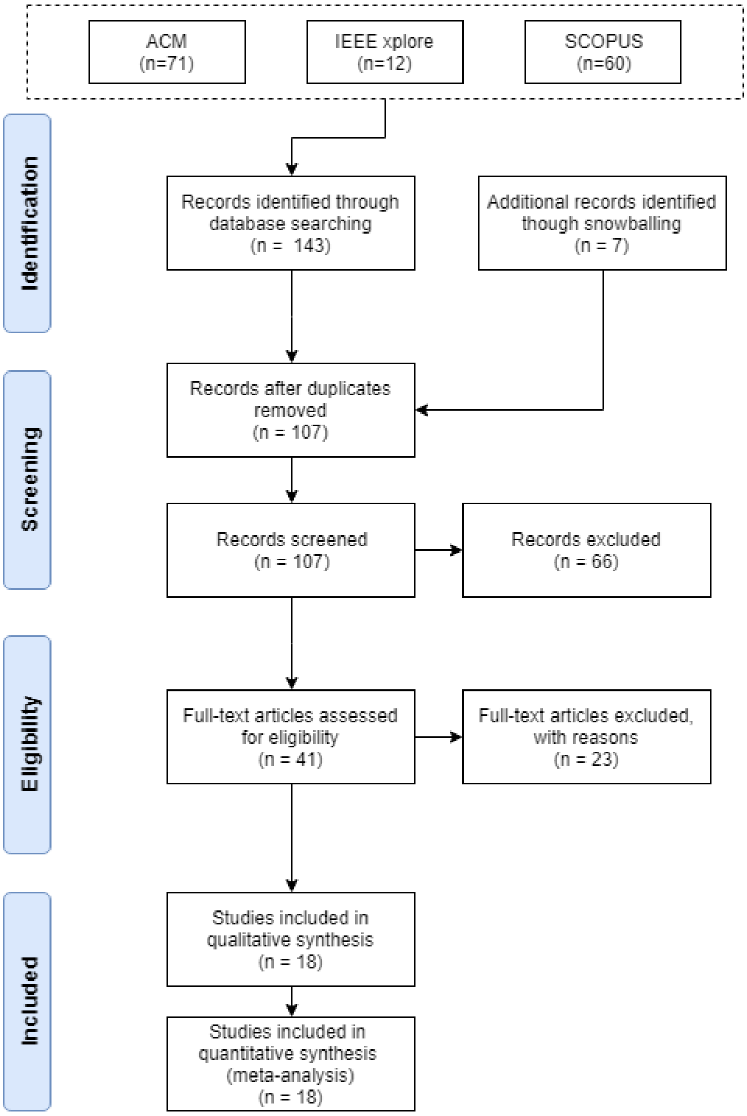

ACM
(n=71)
IEEE xplore
(n=12)
SCOPUS
(n=60)
Identification
Screening
Eligibility
Included
Records identified through database searching (n = 143)
Additional records identified though snowballing (n = 7)
Records after duplicates removed (n = 107)
Records screened (n = 107)
Records excluded (n = 66)
Full-text articles assessed for eligibility (n = 41)
Full-text articles excluded, with reasons (n = 23)
Studies included in qualitative synthesis (n = 18)
Studies included in quantitative synthesis (meta-analysis) (n = 18)

Figure 1. Stages of the study selection process.

3.2. Selection of Primary Studies

To select the suitable 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 [44]. 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 it could be performed with the same terms. The automatic search was complemented by snowballing to select works related to the research question. Duplicated reports were discarded.

Step 2—Read titles and abstracts to identify potentially relevant studies: Potentially relevant studies were identified based on the analysis of their titles and abstracts, discarding studies that were clearly irrelevant to the search. When there was some doubt about whether a study should be included, it was included for consideration at a later stage.

Step 3—Apply inclusion and exclusion criteria: After the previous steps, the sections on introduction, methods, and 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.

Inclusion criteria. IC1: Publications should be “journal” or “conference”. IC2: To be eligible, works should 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 (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 focused only on code smells or on software bugs without any discussion regarding the relationship and/or influence that the first exerts 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. [45].

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?

3.3. 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 complemented by snowballing to identify potentially relevant studies. The snowballing entailed following the references from one paper to another, to find other relevant papers [46]. This was done in both backward and forward directions. Backward snowballing means following the reference list and forward snowballing refers to looking at papers citing the paper that has been found relevant [47]. Next, we applied the inclusion/exclusion criteria to all papers. The first tests using the automatic search began in January 2018. We needed to adapt the search string in some of the repositories presented in Table 2, taking care to preserve its primary meaning and scope. The studies obtained from snowballing were likewise analyzed regarding their titles and abstracts. We tabulated everything on a dataset 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.

3.4. Data Extraction

Based on the results of the selection process described in Section 3.2, we listed and classified the selected primary studies to enable a clear understanding of aims, methodologies, and findings. We 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) analyzed projects; (viii) solution to establish a relationship among bugs and project’s commits; (ix) research questions and respective answers; (x) code smells that have an influence on the occurrence of bugs; (xi) software projects that illustrate this influence.

3.5. Potentially Relevant Studies

Results obtained from both the automatic search and snowballing were included on a dataset. Articles with identical titles, author(s), years and abstracts were considered duplicates and thus discarded. As shown in Figure 1, the search returned a total of 143 references in the automatic search and 7 from snowballing, totaling 150 studies. After reading their titles and abstracts, we deemed 107 papers relevant. We then read the methodology section, the research questions, and corresponding results of 18 articles considered relevant in light of the research questions. Finally, answers to the two research questions—**RQ1** and **RQ2**—were obtained. Table 2 presents an overview of the selection process for the various public data sources. The list of selected papers from this Systematic Literature Review is provided in Tables A1 and A2 in the Appendix A of this paper.

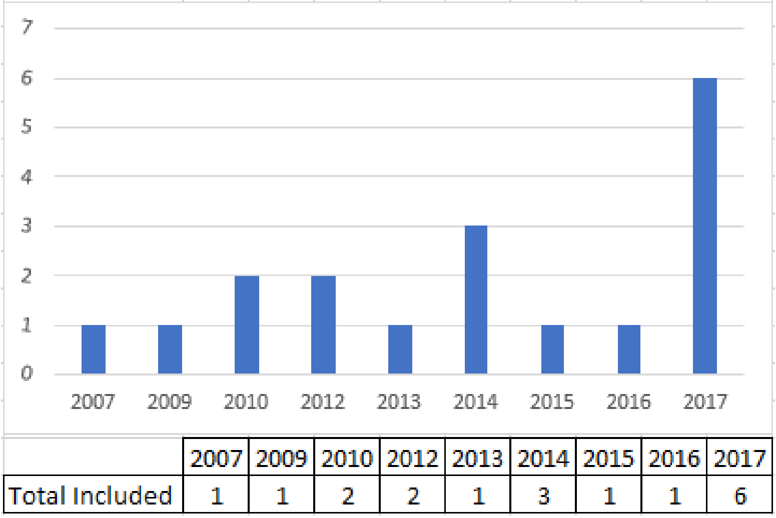
Table 2. Selection process for each public data source.

Public Data Source	Search Result	Relevant Studies	Search Effectiveness
ACM	71	4	5.63%
IEEE	12	2	16.6%
SCOPUS	60	5	8.33%

4. Results and Discussion

This section presents the results of this SLR to answer research questions **RQ1**, **RQ2**. It also outlines the characteristics of the selected studies which are listed in Tables A1 and A2 in the Appendix. The studies are identified with the letter "S", followed by its corresponding number.

Figure 2 depicts the time distribution of the selected studies. It should be noted that from the 18 studies, 6 of them were published in 2017. This could be indicative of a recent surge in interest in this topic on the part of the software engineering community.



	2007	2009	2010	2012	2013	2014	2015	2016	2017
Total Included	1	1	2	2	1	3	1	1	6

Figure 2. Timeline distribution of papers.

4.1. Influence of Code Smells on Software Bugs

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

Table 3 presents how each of the selected studies contributed to answer **RQ1**. In this case, the following 16 studies provide evidence of the influence of code smells on the occurrence of software bugs: S1–S9, S11–S14, S16–S18. On the other hand, two studies—S10 and S15—state that based on the collected evidence, no cause and effect relationship between code smells and occurrence of software bugs should be claimed. In other words, the majority of studies reached the conclusion that classes affected by specific code smells tend to be more change- and fault-prone. In the following paragraphs, we discuss the aforementioned influences.

Table 3. Influence of the code smells on bugs.

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

Figure 3 classifies the code smells according to their influence on software bugs. The most influential code smells are the ones positively associated with error proneness and recognized as possible causative agents of a significant number of software bugs in the projects analyzed. These studies correspond to those whose answer was *Yes* in Table 3. Therefore, studies S10 [10] and S15 [1] were not considered sources for this group of the figure. Note that the *No information* node shown in Figure 3 is not a type of code smell. This node merely indicates that the studies in question did not disclose which code smells had a greater or lesser influence on the bugs of the projects analyzed.

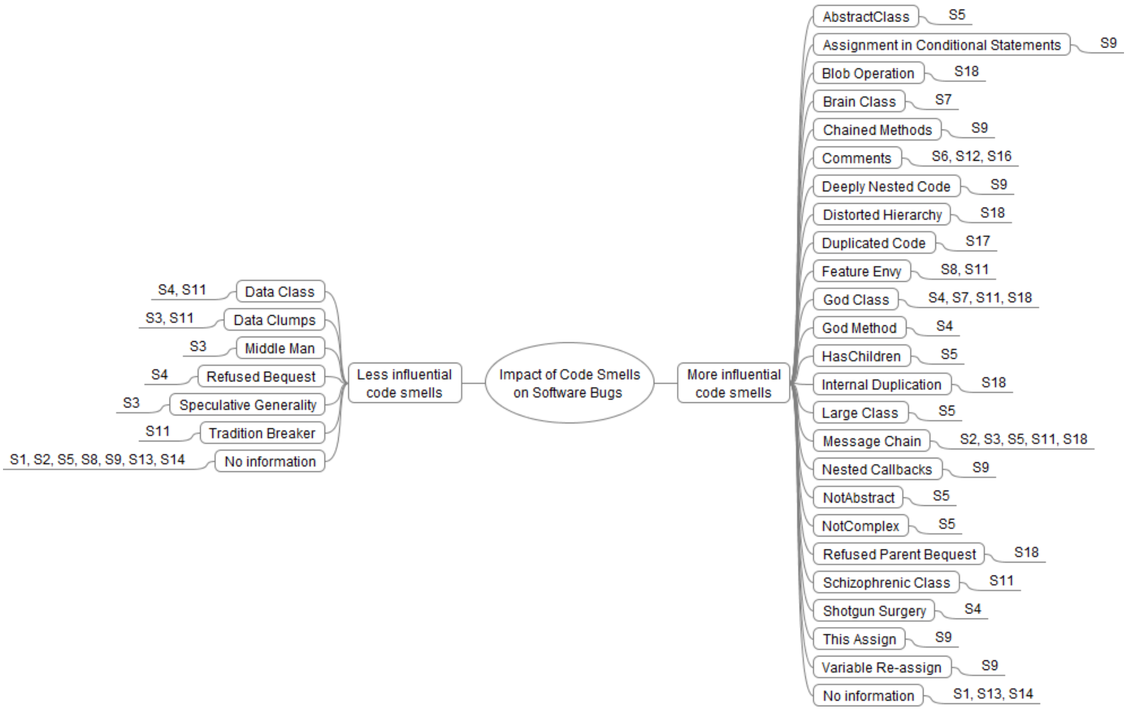


Figure 3. Influential code smells on software bugs.

From the set of problems mentioned in S10 [10], roughly 30% percent is related to files containing code smells. Within the limits established by the context of this study, it is clear that the proportion of problems linked to code smells was not as large as could be expected. For example, the findings in S15 [1] do not support the claim that clones should be approached as a *code smell*. The authors argued that most bugs have very little to do with clones and that even cloned code contains less *buggy* code, i.e., code implicated in bug fixes, than the rest of the code.

The S3 study [8] argued that smell *Middle Man* is related to fewer faults in `ArgoUML` and *Data Clumps* is related to fewer faults in `Eclipse` and `Apache Commons`. They also mentioned that *Data*

Clumps, *Speculative Generality*, and *Middle Man* are associated to a relatively low number of faults in the analyzed software projects. Moreover, the authors report that *Data Clumps*, *Speculative Generality*, and *Middle Man* present a weak relationship with the number of faults in some circumstances and in some systems.

In S4 [9], the authors analyzed the code smells *Data Class*, *God Class*, *God Method*, *Refused Bequest* and *Shotgun Surgery*. They concluded that *God Class* was a significant contributor and was positively associated with error proneness in all releases. *Shotgun Surgery* and *God Method* were significant contributors in Eclipse 2.1 and positively associated with error proneness. In contrast, other code smells seemed to have less influence on bug occurrence. In S11 [13], the code smells *Tradition Breaker*, *Data Clumps* and *Data Class* had the lowest proportions of bugs in their classes with percentages smaller than or equal to 5%.

In S1 [5], the authors argued that class fault proneness is significantly higher in classes in which co-occurrence of anti-patterns and clones is found, in comparison with other classes in the analyzed software projects. S2 [7] also claimed that classes with code smells tend to be more change- and fault-prone than other classes and that this is even more noticeable when classes are affected by multiple smells [7]. The authors reported high fault-proneness for the code smell *Message Chains*. Of the 395 releases analyzed in 30 projects, *Message Chains* 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 that although *Message Chains* is potentially harmful, its diffusion is rather limited. S3 emphasized that *Message Chains* increased the flaws in two software projects (`Eclipse` and `ArgoUML`). However, the authors note that when detected in larger files, the number of detected flaws relative to file size was actually smaller.

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 of these smells on faults seems to be small [8]. The authors argued that, in general, the smells analyzed have a relatively small influence (always below 10%) on bugs except for *Message Chains* and suggested that *Switch Statements* had no effect in any of the three projects analyzed. The authors argued that *Data Clumps* reduced the failures in `Apache` and `Eclipse` but increased the number of failures in `ArgoUML`. *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 on bugs of smells *Shotgun Surgery*, *God Class*, and *God Method*. The authors analyzed the post-release system evolution process to reach this conclusion [9]. Results confirmed that some 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 suggests that code smells could be used as a systematic method to identify 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 to represent 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 a relevant relationship between smells *Data Class* and *Refused Bequest* and the occurrence of bugs.

Although not explicitly investigating the occurrence of bugs as a consequence of code smells, S5 [37] analyzed the influence of a set of code smells in class change-proneness. The study took into account a plethora of previous research findings to argue that change-proneness increases the probability of the advent of bugs in software projects [48–51]. Considering this scenario, we included study S5 [37] in this SLR. Twenty-nine code smells were analyzed in this study throughout nine releases of the `Azureus` software project and in thirteen releases of the `Eclipse` software project.

The authors analyzed these releases to understand to what extent these code smells influenced class change-proneness. The conclusion was that, in almost all releases of projects `Azureus` and `Eclipse`, classes with code smells were more change-prone than other classes, and specific smells were more correlated to change-proneness than other smells [37]. In the case of the `Azureus`, the smell *Not Abstract* had a significant impact on change proneness in more than 75% of releases, whereas *Abstract Class* and *Large Class* proved to be relevant as agents of change-proneness in more than 50% of the analyzed releases. In project `Eclipse`, *HasChildren*, *MessageChainsClass*, and *NotComplex* smells had a significant effect on change-proneness for 75% of the releases or more.

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

S9 [39] reported that empirical evidence suggests that `JavaScript` files without code smells have 65% lower hazard rates 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 have smells. 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 smells *Schizophrenic Class* and *Tradition Breaker* on the occurrence of bugs. *Schizophrenic Class* was associated to a significant proportion of bugs, whereas *Tradition Breaker* seemed to have a low influence. The authors recommended investigating these smells in other systems with an emphasis on *Schizophrenic Class*. Still, according to S11 [13], empirical evidence from project `Apache Ant` indicated that classes affected by code smells are up to three times more likely to show bugs than other classes. In the case of the `Apache Xerces` software project, the odds rate proved to be up to two times more likely to show bugs.

S13 [52] 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 projects to fix the inefficient usage of regular expressions, recurrent computations of constant data, and usage of deprecated decryption algorithms.

S14 [53] comprised an analysis of 34 software projects and reported a significant positive correlation between number of bugs and number of anti-patterns. It also reported a negative correlation between number of anti-patterns and maintainability. This further supports the intuitive thinking that establishes a relation between anti-patterns, bugs, and (lack of) quality.

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 with merge conflicts. From those, *God Class*, *Message Chains*, *Internal Duplication*, *Distorted Hierarchy*, and *Refused Parent Bequest* stood out. The only two (significant) smells associated with semantic conflicts in the S18 study are *Blob Operation* and *Internal Duplication*, which proved to be respectively 1.77 and 1.55 times more likely to be present in a semantic conflict than with non-semantic conflicts.

Figure 3 shows that *God Class* stood out from the other smells in 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 3 releases of project `Eclipse`. Likewise, *Message Chains* also stood out as an influential factor, as reported in studies S2, S3, S5, S11, and S18. On the other hand, no relevant tendency to change- and fault-prone was reported as a consequence of smells *Data Class*, *Refused Bequest*, *No information*, *Tradition Breaker*, *Data Clumps*, *Middle Man*, and *Switch Statements*. As can be seen in the same figure, a node labeled *No information* is connected to *Less Influential Code*

Smells. In this case, studies S1, S2, S5, S8, S9, S13, and S14 did not provide information on which 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 Chains* 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 projects `Eclipse` and `ArgoUML`, though no influence was detected in 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 project `Azureus`. *AbstractClass* and *LargeClass* proved influential to a significant degree on more than 50% of the releases (five out of nine) of project `Azureus`, according to S5. In `Eclipse`, the smells *HasChildren*, *MessageChainsClass*, and *NotComplex* had a significant influence on the change-proneness in more of 75% of the releases. In S9, the risk rate varied between the 5 projects analyzed, since smells *Chained Methods*, *This Assign* and *Variable Re-assign* had the highest hazard ratios in project `Express`. Smells *Nested Callbacks*, *Assignment in Conditional Statements* and *Variable Re-assign* had the highest hazard rates in project `Grunt`. *Deeply Nested Code* was the most hazardous smell in terms of bug influence in project `Bower`. In terms of bug influence, *Assignment in Conditional Statements* had the highest hazard ratio in project `Less.js` and *Variable Re-assign* had the highest hazard ratio in project `Request`.

Four studies focused on one specific smell: S6, S12, S16 on *Comments* and S17 on *Code Clone*. Their authors concluded that they have an influence on bug occurrence in the projects analyzed and therefore they were considered more influential. S1, S13, S14 did not disclose which smells exerted the greatest influence on bug occurrence. S1, S2, S5, S8, S9, S13, S14 did not provide information on which code smells were less influential.

S6, S12, and S16 analyzed the *Code Comments* smell. The S12 results suggest there is 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 contributed to improve comprehensibility, but often the motivation for comments can be to compensate for a lack of comprehensibility of complicated and difficult-to-understand code. Consequently, some well-written comments may be indicative of low-quality code. For S6, methods with more comments than the quantity estimated on the basis of its size and complexity are about 1.6–2.8 times more likely than average to be faulty. 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, and 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 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 in general, code smells are only partial indicators of maintenance difficulties, because the study results showed a relatively low coverage of smells when observing project maintenance. 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 those. The S10 study suggests that to evaluate code more widely and safely, different analysis techniques should be combined with code smell detection.

The results of S15 are 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 did 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 by the number of lines changed to fix a bug). Most of the bugs (more than 80%) had no cloned code and around 90% of bugs yielded a clone ratio lower than the average project. In other words, S10 and S15 diverge from the other selected studies in that they did not find evidence of a clear relationship between code smells and software bugs.

4.2. 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 were used to find evidence of the influence of code smells on the occurrence of software bugs?* Evidence collected from selected studies is presented in Figure 4.

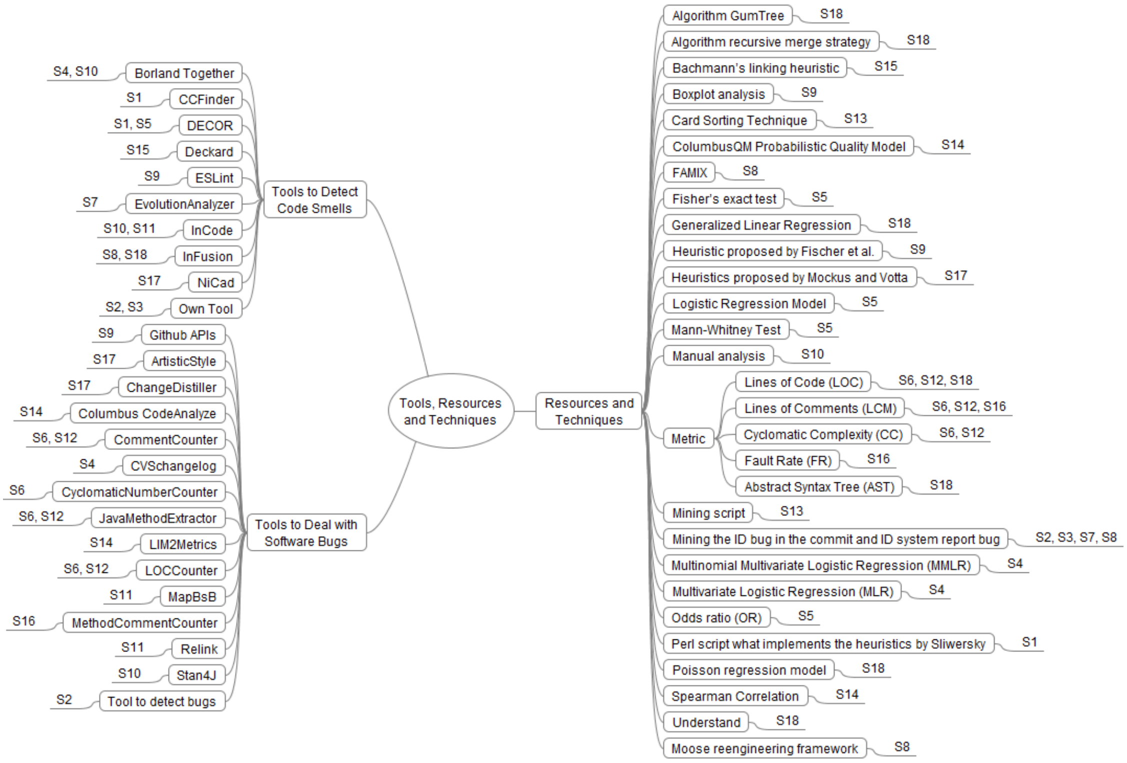


Figure 4. Tools, resources, and techniques to identify the influence of specific code smells on bugs.

Figure 4 depicts tools, resources, and techniques reported in the selected studies to investigate the influence of specific smells on software bugs. S1 evaluated fault-proneness by analyzing the change and fault reports included in the software repositories. S1 used the version-control systems `CVS` (http://cvs.nongnu.org/), also used by S5 and S8, and `Subversion` (http://subversion.apache.org/), also used in studies S7, S10, S11, and S17. S1 executed a `Perl` script that implemented the heuristics proposed in [54] to analyze the commit 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 was then compared with the project 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 [55], which checks whether a proportion varies between two samples and also computes the odds ratio (OR) [55] 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 [56] was used, similarly to Vokac's study [57].

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, S7 for some projects, used the Jira tracking system. S15 used Git. S2, S3, S9, S14, S16, and 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, the tool detection rules are generally more restrictive to ensure a good compromise between recall and precision, with the consequence that they may have missed some smell instances. To validate this claim, they evaluated the behavior of three existing tools: DECOR, JDeodorant, and HIST. S15 used Deckard [58]. File name, line number, information about which clone a line belongs to and sibling clones were extracted from the Deckard output. S8 parsed the source code with the tool inFusion (http://www.intooitus.com/inFusion.html) and produced a FAMIX (FAMIX is a language-independent, object-oriented meta-model of the software system presented in [59], whose results are a list of method-level design flaws from each class). Having the FAMIX-compliant model as input, the authors used detection strategies to spot the design flaws. This operation was performed within the Moose reengineering framework [60], and the result was a list of method level design flaws within each class.

To identify a relationship between bugs and source code, S2, S3, S8 resorted to the use of 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 the issue tracking system's issue identifier [61,62]. In S8, some false positives were derived when the approach was limited to simply looking a the number. In their algorithm, each time a candidate reference to a bug report is found, a check is made to confirm that a bug with such an ID indeed exists. A check is also made, that the date in which the bug was reported is indeed prior to the timestamp of the commit comment in which the reference was found (i.e., it checks that the bug is fixed *after* it is reported).

S4 and S10 used Borland Together (http://www.borland.com/us/products/together)—a plug-in tool for Eclipse—to identify the classes that had bad smells. S10 also used InCode (http://www.intooitus.com/products/incode). To analyse the change reports from the Eclipse change log, S4 used CVSchangelog, a tool available at Sourceforge. S4 used two types of dependent variable 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 (http://trac.edgewall.org), 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 (http://www.intooitus.com/products/incode). Files considered problematic but 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 evidence 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` (http://stan4j.com) were used to identify such couplings.

S6, S12, and S16 comprised empirical analyses 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 analyzing 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 at low cost. S16 did not disclose the control system used. The failures of the projects analysed in S12 were 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) to obtain the number of comments in a method. Then, they collected the change 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 built the complete list of methods except for the abstract methods. They used `JavaMethodExtractor` to extract the method data. For each method, they collected its change history. They examined whether the method was changed by checking all commits corresponding to the source file the target method is declared in. A method was deemed faulty if one of its changes was a bug fixing.

For data collection, S6 used `JavaMethodExtractor`, `CommentCounter`, `LOCCounter`, and `CyclomaticNumberCounter` and obtained the latest version of the source files from the repository, and made a list of all methods in the files, except for abstract methods. Then, the "initial" versions of the methods were also obtained by tracing their change history, and the LCM, LOC, and CC values were taken from those initial versions. For each method, they checked whether the method was faulty by examining all changes in which the method was involved. S6 and S12 classified a change as a bug fix when the corresponding commit's log included one of the bug-fix-related words—"bug, fix, defect"—while S16 used the *Lines of Comments* (LCM) metric to compute the number of comments written in a module and the FR metric [63] to estimate the fault-proneness of modules.

S9 implemented all the steps of their approach to detect code smells in `JavaScript` into a framework available on `GitHub` (https://GitHub.com/amir-s/smelljs). 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 used to perform an analysis at the commit level. S9 used `GitHub` APIs to obtain the list of all resolved issues on the systems. They leveraged the *SZZ* algorithm [54] to detect changes that introduced faults. Fault-fixing commits were identified using the heuristic proposed by Fischer et al. [61], which comprises the use of 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 extracted 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 the corresponding dates of bug creation. To automatically detect code smells in the source code, the *Abstract Syntax Tree* (AST) was first extracted from the code, using `ESLint` (http://eslint.org/), 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 to compare the time until a fault occurred 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 [64].

The goal of S13 was to investigate performance-related commits in Android apps with the purpose of understanding their nature and their relationship with project characteristics, such as domain and size. The study target in 2443 open-source apps taken from Google Play store considering their releases 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 only considers the folder containing the source code and resources of the mobile app, excluding back end, documentation, tests, 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, minor, stuck, instant, respons, react, speed, latenc, perform, throughput, hang, memory, or leak. These keywords were 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 [65] 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 informative titles (e.g., UI issues, file system issues).

To ensure a quality assessment, S14 chose the *ColumbusQM* probabilistic quality model [66] 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 the Columbus CodeAnalyzer reverse engineering tool [67]. S14 compiled the types of data described above for a total of 228 open-source Java systems, 34 of which had 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, LIM2Metrics was used to compute various 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 was somewhat masked by the ranking—which can be viewed as a kind of data loss—but this is not too important as they were 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 for bad smell detection (Intooitus, the company that provided InCode, seems to have closed and the tool website is no longer available). To help with information processing, 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 were processed by the Relink tool (available at https://code.google.com/archive/p/bugcenter/wikis/ReLink.wiki), which, in turn, generated a file with bugs and associated revisions (commits). Finally, bad smells, bug-revisions, and change log files were processed so that software versions, link bugs to system classes and versions, and cross related information between bad smells and bugs could be analyzed.

S17 extracted the SVN commit messages by applying SVN log command to identify bug-fix commits of candidate systems and apply the heuristics proposed by Mockus and Votta [68] on the commit messages to automatically identify bug-fix commits. They analyzed and identified all the cloned methods relating to bug-fixes, and analyzed 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 ArtisticStyle (https://sourceforge.net/projects/astyle/) and extracted 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 extracted method information from the successive revisions of the source code. They stored the method information (file path, package name, class name, signature, start line, end line) in a database for mapping changes to the corresponding methods.

S18 selected active open source projects hosted on GitHub that used 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 most likely to used by the average Git project. They used GumTree for their analysis, as it allowed them to track elements at the AST level. This way, they tracked just the elements that they were interested in (statements) and ignored other changes that do not actually change the code. The assessment of the impact of code smells was based on the number of bug-fixes occurred on code lines associated with a given code smell and a merge conflict. The authors used *Generalized Linear Regression* where the dependent variable (count of bug fixes occurring on smelly and conflicting lines) follows a *Poisson* distribution. Therefore, they used a *Poisson* regression model with a log linking function. They counted the number of references to and from other files to the files that were involved in a conflict. They also collected other factors for each commit such as the difference between the two merged branches in terms of LoC and 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 a VIF score threshold of 5.

4.3. Software Projects Analyzed 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 Java language, except those from studies S15 (C) and S9 (JavaScript). Moreover, it was possible to identify the names of all projects, except the four projects reported in study S10, referenced as software projects A, B, C, and D.

Table 4. Software projects analyzed in the selected studies.

Project	Language	Study
Eclipse	Java	S1, S3, S4, S5, S16
Apache Xerces	Java	S2, S7, S11
Apache Ant	Java	S2, S11, S16
Apache Lucene	Java	S2, S7, S8
Apache Tomcat	Java	S11, S16
ArgoUML	Java	S2, S3
Azureus	Java	S1, S5
Checkstyle Plug-in; PMD; SQuirreL SQL Client; Hibernate ORM	Java	S12, S6
JHotDraw	Java	S1, S2
Apache Commons	Java	S3
Apache Jmeter; Apache Lenya	Java	S11
Apache httpd; Nautilus; Evolution; GIMP	C	S15
Software Project A; Software Project B; Software Project C; Software Project D (undisclosed names)	Java	S10
Request; Less.js; Bower; Express; Grunt	JavaScript	S9
Log4J	Java	S7
Maven; Mina; Eclipse CDT; Eclipse PDE UI; Equinox	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	Java	S2
DNSJava; JabRef; Carol; Ant-Contrib; OpenYMSG	Java	S17
2443 apps (undisclosed names)	Not provided by the authors	S13
34 projects (undisclosed names)	Java	S14
143 projects (undisclosed names)	Java	S18

5. 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 answer 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. In the present work, it was addressed by specifying a research protocol that defines the research questions and objectives of the study, inclusion and exclusion criteria, search strings to be used, 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, and hence, their influences on software bugs, could have been detected. The goal was 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.

6. Conclusions

This study aims to provide a detailed panorama of the state-of-the-art analyses on how code smells influence the occurrence of bugs in software projects, as well as tools, resources, and techniques that support the analysis. To achieve this goal, we performed a systematic literature review and investigated studies published in top software engineering venues. The results of the study suggest that under certain circumstances, specific code smells do have an influence on the occurrence of software bugs. We considered evidence provided by 18 studies published in three digital libraries enabling us to answer the two research questions of this SLR. For **RQ1**, of the 18 studies analysed, 16 presented evidence that code smells do have an influence on occurrence of bugs. Some studies highlighted that code smells exert a greater or lesser influence on the bugs and only two concluded that the code smells did not have an influence on the occurrence of bugs. For **RQ2**, we identified tools that detect code smells, and complementary tools that can be used in the analyses, 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 to bugs registered in the portals.

The evidence gathered in the results of the selected studies showed that specific types of code smells do influence occurrence of software bugs. In other words, classes affected by code smells are more prone to changes and to contain failures than classes that are not affected by smells.

The vast majority of the selected studies did not consider the characteristics of the bugs that were probably influenced by classes affected by the aforementioned code smells. As future work, we plan to analyze the influence of code smells on different types of bugs, for example the relationship of bugs related to security and their associations with specific code smells and how these bugs have been classified by the software project community in terms of severity and priority.

Author Contributions: A.S.C. and G.d.F.C. together searched for eligible papers from the publication databases and read the eligible papers carefully. A.S.C., G.d.F.C., and M.P.M. wrote the article. All authors read and approved the final manuscript.

Funding: This research received no external funding.

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

Appendix A

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

Table A1. List of selected studies from the search string

ID	Author, Title	Venue	Year
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
S16 [69]	Aman, H. *An Empirical Analysis on Fault-proneness of Well-Commented Modules*	IWESEP	2012
S15 [1]	Rahman, F.; Bird, C.; Devanbu, P. *Clones: What is that smell?*	MSR	2012
S14 [53]	Bán, D.; Ferenc, R. *Recognizing antipatterns and analyzing their effects on software maintainability*	ICCSA	2014
S3 [8]	Hall, T.; Zhang, M.; Bowes, D.; Sun, Y. *Some code smells have a significant but small effect on faults*	ATSME	2014
S12 [63]	Aman, H.; Amasaki, S.; Sasaki, T.; Kawahara, M. *Empirical Analysis of Fault-Proneness in Methods by Focusing on their Comment Lines*	APSEC	2014
S13 [52]	Das, T.; Di Penta, M.; Malavolta, I. *A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps*	ICSME	2016
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
S11 [13]	Nascimento, R.; Sant'Anna, C. *Investigating the Relationship Between Bad Smells and Bugs in Software Systems*	SBCARS	2017
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 A2. List of selected studies from snowballing.

ID	Author, Title	Forward/Backward Snowballing	Venue	Year
S5 [37]	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
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
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
S6 [70]	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
S9 [39]	Saboury, A.; Musavi, P.; Khomh, F.; Antoniol, G. *An Empirical Study of Code Smells in `JavaScript` projects*	Forward of S2	SANER	2017
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

References

1. Rahman, F.; Bird, C.; Devanbu, P. Clones: What is that smell? *Empir. Softw. Eng.* **2012**, *17*, 503–530. [CrossRef]
2. Fowler, M.; Beck, K. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley Professional: Boston, MA, USA, 1999.
3. D'Ambros, M.; Bacchelli, A.; Lanza, M. On the impact of design flaws on software defects. In Proceedings of the 10th International Conference on Quality Software (QSIC), Los Alamitos, CA, USA, 14–15 July 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. In Proceedings of the 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), Shanghai, China, 17–18 September 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. In Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, 25–29 July 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. In Proceedings of the 16th Working Conference on Reverse Engineering, Lille, France, 13–16 October 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. *Empir. Softw. Eng.* **2017**, *23*, 1188–1221. [CrossRef]
8. Hall, T.; Zhang, M.; Bowes, D.; Sun, Y. Some code smells have a significant but small effect on faults. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2014**, *23*, 33. [CrossRef]
9. Li, W.; Shatnawi, R. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *J. Syst. Softw.* **2007**, *80*, 1120–1128. [CrossRef]
10. Yamashita, A.; Moonen, L. To what extent can maintenance problems be predicted by code smell detection?—An empirical study. *Inf. Softw. Technol.* **2013**, *55*, 2223–2242. [CrossRef]
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. In Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM), Timisoara, Romania, 12–18 September 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. In Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Toronto, ON, Canada, 9–10 November 2017; pp. 58–67.
13. Nascimento, R.; Sant'Anna, C. Investigating the relationship between bad smells and bugs in software systems. In Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse, Fortaleza, Ceará, Brazilm, 18–19 September 2017; p. 4.
14. Rasool, G.; Arshad, Z. A review of code smell mining techniques. *J. Softw. Evol. Process* **2015**, *27*, 867–895. [CrossRef]
15. Fernandes, E.; Oliveira, J.; Vale, G.; Paiva, T.; Figueiredo, E. A review-based comparative study of bad smell detection tools. In Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, Limerick, Ireland, 1–3 June 2016; p. 18.
16. Fontana, F.A.; Braione, P.; Zanoni, M. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.* **2012**, *11*, 1–38.
17. Lehman, M.M. Laws of software evolution revisited. In *European Workshop on Software Process Technology*; Springer: Berlin, Germany, 1996; pp. 108–124.
18. 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. In Proceedings of the 37th International Conference on Software Engineering-Volume 1, Toronto, ON, Canada, 9–10 November 2017; pp. 403–414.
19. Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; Poshyvanyk, D.; De Lucia, A. Mining version histories for detecting code smells. *IEEE Trans. Softw. Eng.* **2015**, *41*, 462–489. [CrossRef]
20. 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 Trans. Softw. Eng.* **2010**, *36*, 20–36. [CrossRef]
21. Tsantalis, N.; Chatzigeorgiou, A. Identification of move method refactoring opportunities. *IEEE Trans. Softw. Eng.* **2009**, *35*, 347–367. [CrossRef]

22. Sahin, D.; Kessentini, M.; Bechikh, S.; Deb, K. Code-smell detection as a bilevel problem. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2014**, *24*, 6. [CrossRef]
23. Khomh, F.; Vaucher, S.; Guéhéneuc, Y.G.; Sahraoui, H. A bayesian approach for the detection of code and design smells. In Proceedings of the 9th International Conference on Quality Software, Jeju, South Korea, 24–25 August 2009; pp. 305–314.
24. Marinescu, R. Detection strategies: Metrics-based rules for detecting design flaws. In Proceedings of the 20th IEEE International Conference on Software Maintenance, Chicago, IL, USA, 11–14 September 2004; pp. 350–359.
25. Kessentini, W.; Kessentini, M.; Sahraoui, H.; Bechikh, S.; Ouni, A. A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Trans. Softw. Eng.* **2014**, *40*, 841–861. [CrossRef]
26. Danphitsanuphan, P.; Suwantada, T. Code smell detecting tool and code smell-structure bug relationship. In Proceedings of the 2012 Spring Congress on Engineering and Technology (S-CET), Xi'an, China, 27–30 May 2012; pp. 1–5.
27. Griswold, W.G. *Program Restructuring as an Aid to Software Maintenance*; University of Washington: Seattle, WA, USA, 1992.
28. Griswold, W.G.; Notkin, D. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **1993**, *2*, 228–269. [CrossRef]
29. Opdyke, W.F. Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems. Available online: https://refactory.com/papers/doc_details/29-refactoring-an-aid-in-designing-application-frameworks-and-evolving-object-oriented-systems (accessed on 6 November 2018).
30. Opdyke, W.F. *Refactoring Object-Oriented Frameworks*; University of Illinois at Urbana-Champaign: Champaign, IL, USA, 1992.
31. Ambler, S.; Nalbone, J.; Vizdos, M. *Enterprise Unified Process, The: Extending the Rational Unified Process*; Prentice Hall Press: Upper Saddle River, NJ, USA, 2005.
32. Van Emden, E.; Moonen, L. Java quality assurance by detecting code smells. In Proceedings of the 2002 Ninth Working Conference on Reverse Engineering, Richmond, VA, USA, 29 October–1 November 2002; pp. 97–106.
33. 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, Santa Barbara, CA, USA, 29 July–3 August 2001; pp. 173–182.
34. Marinescu, R. Measurement and quality in object-oriented design. In Proceedings of the 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, 26–29 September 2005; pp. 701–704.
35. 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: Berlin, Germany, 2007.
36. Mantyla, M.; Vanhanen, J.; Lassenius, C. A taxonomy and an initial empirical study of bad smells in code. In Proceedings of the 2003 International Conference on Software Maintenance, Amsterdam, The Netherlands, 22–26 September 2003; pp. 381–384.
37. Khomh, F.; Di Penta, M.; Guéhéneuc, Y.G.; Antoniol, G. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empir. Softw. Eng.* **2012**, *17*, 243–275. [CrossRef]
38. 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.: Newton, MA, USA, 2016.
39. Saboury, A.; Musavi, P.; Khomh, F.; Antoniol, G. An empirical study of code smells in javascript projects. In Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 21–24 February 2017; pp. 294–305.
40. Yamashita, A.; Moonen, L. Do developers care about code smells? An exploratory survey. In Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), Koblenz, Germany, 14–17 October 2013; pp. 242–251.
41. Olbrich, S.; Cruzes, D.S.; Basili, V.; Zazworka, N. The evolution and impact of code smells: A case study of two open source systems. In Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, Lake Buena Vista, FL, USA, 15–16 October 2009; pp. 390–400.

42. Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; De Lucia, A.; Poshyvanyk, D. Detecting bad smells in source code using change history information. In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, Silicon Valley, CA, USA, 11–15 November 2013; pp. 268–278.
43. Kitchenham, B.; Charters, S. *Guidelines for performing Systematic Literature Reviews in Software Engineering*; EBSE: Durham, UK, 2007; Volume 2.
44. Zhang, H.; Babar, M.A.; Tell, P. Identifying relevant studies in software engineering. *Inf. Softw. Technol.* **2011**, *53*, 625–637. [CrossRef]
45. Dybå, T.; Dingsøyr, T. Empirical studies of agile software development: A systematic review. *Inf. Softw. Technol.* **2008**, *50*, 833–859. [CrossRef]
46. Skoglund, M.; Runeson, P. Reference-based search strategies in systematic reviews. In Proceedings of the 13th international conference on Evaluation and Assessment in Software Engineering, Swindon, UK, 20–21 April 2009.
47. Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslén, A. *Experimentation in Software Engineering*; Springer Science & Business Media: Berlin, Germany, 2012.
48. Cotroneo, D.; Pietrantuono, R.; Russo, S.; Trivedi, K. How do bugs surface? A comprehensive study on the characteristics of software bugs manifestation. *J. Syst. Softw.* **2016**, *113*, 27–43. [CrossRef]
49. Qin, F.; Zheng, Z.; Li, X.; Qiao, Y.; Trivedi, K.S. An empirical investigation of fault triggers in android operating system. In Proceedings of the 22nd Pacific Rim International Symposium on Dependable Computing (PRDC), Christchurch, New Zealand, 22–25 January 2017; pp. 135–144.
50. González-Barahona, J.M.; Robles, G. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empir. Softw. Eng.* **2012**, *17*, 75–89. [CrossRef]
51. Panjer, L.D. Predicting eclipse bug lifetimes. In Proceedings of the Fourth International Workshop on Mining Software Repositories, Minneapolis, MN, USA, 20–26 May 2007; p. 29.
52. Das, T.; Di Penta, M.; Malavolta, I. A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps. In Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), Raleigh, NC, USA, 2–7 October 2016; pp. 443–447.
53. Bán, D.; Ferenc, R. Recognizing antipatterns and analyzing their effects on software maintainability. In *Computational Science and Its Applications—ICCSA 2014*; Springer: Cham, Switzerland, 2014; pp. 337–352.
54. Śliwerski, J.; Zimmermann, T.; Zeller, A. When do changes induce fixes? In Proceedings of the 2005 International Workshop on Mining Software Repositories, St. Louis, MO, USA, 17 May 2005; ACM Sigsoft Software Engineering Notes. ACM: New York, NY, USA, 2005; Volume 30, pp. 1–5.
55. Sheskin, D.J. *Handbook of Parametric and Nonparametric Statistical Procedures*; CRC Press: Boca Raton, FL, USA, 2003.
56. Hosmer, D.; Lemeshow, S. *Applied Logistic Regression*, 2nd ed.; John Wiley & Sons: New York, NY, USA, 2000.
57. Vokac, M. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Trans. Softw. Eng.* **2004**, *30*, 904–917. [CrossRef]
58. Jiang, L.; Misherghi, G.; Su, Z.; Glondu, S. Deckard: Scalable and accurate tree-based detection of code clones. In Proceedings of the 29th international conference on Software Engineering, Minneapolis, MN, USA, 20–26 May 2007; pp. 96–105.
59. Demeyer, S.; Tichelaar, S.; Ducasse, S. *FAMIX 2.1—The FAMOOS Information Exchange Model*; Technical Report; University of Bern: Bern, Switzerland, 2001.
60. Nierstrasz, O.; Ducasse, S.; Gı̌rba, T. The story of Moose: An agile reengineering environment. *ACM SIGSOFT Softw. Eng. Notes* **2005**, *30*, 1–10. [CrossRef]
61. Fischer, M.; Pinzger, M.; Gall, H. Populating a release history database from version control and bug tracking systems. In Proceedings of the International Conference on Software Maintenance, Amsterdam, The Netherlands, 22–26 September 2003; pp. 23–32.
62. Čubranić, D.; Murphy, G.C. Hipikat: Recommending pertinent software development artifacts. In Proceedings of the 25th international Conference on Software Engineering, Portland, OR, USA, 3–10 May 2003; pp. 408–418.
63. Aman, H. Quantitative analysis of relationships among comment description, comment out and fault-proneness in open source software. *IPSJ J.* **2012**, *53*, 612–621.
64. Fox, J.; Weisberg, S. *An R Companion to Applied Regression*; Sage Publications: Thousand Oaks, CA, USA, 2010.
65. Spencer, D. *Card Sorting: Designing Usable Categories*; Rosenfeld Media: New York, NY, USA, 2009.

66. Bakota, T.; Hegedűs, P.; Körtvélyesi, P.; Ferenc, R.; Gyimóthy, T. A probabilistic software quality model. In Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM), Williamsburg, VI, USA, 25–30 September 2011; pp. 243–252.
67. Ferenc, R.; Beszédes, Á.; Tarkiainen, M.; Gyimóthy, T. Columbus-reverse engineering tool and schema for C++. In Proceedings of the International Conference on Software Maintenance, Washington, DC, USA, 3–6 October 2002; pp. 172–181.
68. Mockus, A.; Votta, L.G. *Identifying Reasons for Software Changes Using Historic Databases*; ICSM: Daejeon, Korea, 2000; pp. 120–130.
69. Chen, J.C.; Huang, S.J. An empirical analysis of the impact of software development problem factors on software maintainability. *J. Syst. Softw.* **2009**, *82*, 981–992. [CrossRef]
70. Aman, H.; Amasaki, S.; Sasaki, T.; Kawahara, M. Lines of comments as a noteworthy metric for analyzing fault-proneness in methods. *IEICE Trans. Inf. Syst.* **2015**, *98*, 2218–2228. [CrossRef]

© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).