Preprint of a chapter: Paweł Piotrowski and Lech Madeyski, "Software defect prediction using bad code smells: A systematic literature review," in Data-Centric Business and Applications: Towards Software Development (Volume 4) (A. Poniszewska-Maranda, N. Kryvinska, S. Jarzabek, and L. Madeyski, eds.), vol. 40 of book series Lecture Notes on Data Engineering and Communications Technologies, pp. 77–99, Cham: Springer International Publishing, 2020. DOI: 10.1007/978-3-030-34706-2_5 [BibTeX]
Draft: http://madeyski.e-informatyka.pl/download/PiotrowskiMadeyski20LNDECT.pdf

Software defect prediction using bad code smells: A systematic literature review

Paweł Piotrowski (✉) and Lech Madeyski

Abstract The challenge of effective refactoring in the software development cycle brought forward the need to develop automated defect prediction models. Among many existing indicators of bad code, *code smells* have attracted particular interest of both the research community and practitioners in recent years. In this paper, we describe the current state-of-the-art in the field of bug prediction with the use of code smells and attempt to identify areas requiring further research. To achieve this goal, we conducted a systematic literature review of 27 research papers published between 2006 and 2019. For each paper, we (i) analysed the reported relationship between *smelliness* and *bugginess*, as well as (ii) evaluated the performance of code smell data used as a defect predictor in models developed using machine learning techniques. Our investigation confirms that code smells are both positively correlated with software defects and can positively influence the performance of fault detection models. However, not all types of smells and smell-related metrics are equally useful. *God Class*, *God Method*, *Message Chains* smells and *Smell intensity* metric stand out as particularly effective. Smells such as *Inappropriate Intimacy*, *Variable Re-assign*, *Clones*, *Middle Man* or *Speculative Generality* require further research to confirm their contribution. Metrics describing the introduction and evolution of anti-patterns in code present a promising opportunity for experimentation.

Paweł Piotrowski
Faculty of Computer Science and Management
Wroclaw University of Science and Technology
Wroclaw, Poland
e-mail: pawel.piotrowski@student.pwr.edu.pl

Lech Madeyski
Faculty of Computer Science and Management
Wroclaw University of Science and Technology
Wroclaw, Poland
ORCiD: 0000-0003-3907-3357
e-mail: lech.madeyski@pwr.edu.pl

1 Introduction

Maintenance constitutes a substantial part of every software development cycle. To ensure the effectiveness of that process, a need arises to guide the maintenance and refactoring effort in a way that ensures most fault-prone components are given a priority. Among many possible types of indicators of such ”hazardous” code, code smells have attracted a lot of interest from the scientific community, as well as practitioners, in the recent years.

The notion of code smells has first been introduced by Fowler et al. [5]. They defined a set of 22 common exemplifications of bad coding practices, that can potentially signal a need for refactoring. Since then, numerous papers have been published on code smells definitions, code smell detection techniques and code smell correlation with software bugs. Some studies on software defect prediction models built with the use of machine learning techniques also attempted to evaluate the effectiveness of code smell information as a bug predictor.

A few systematic literature reviews exist regarding the topic of code smells. Freitas [6] performed a comprehensive review of the ”code smell effect” to assess the usefulness of code smells as a concept. Azeem [3] reviewed machine learning techniques of code smell detection. However, an initial review of related work indicated that no exhaustive review concerning the code smells relationship with software defects and their usefulness as bug predictors have been conducted. In this paper, we attempt to perform such an investigation.

The main goals of this review are:

- to assess the state of the art in the field of fault prediction models that include code smell information,
- to evaluate the contribution of individual smell-related factors,
- to identify promising fields for further research in this area.

The rest of this paper is structured as follows. In Section 2 we describe related studies we were familiar with prior to conducting this review. Section 3 presents the research methodology, including research questions, search strategy and our approach to the process of study selection, quality assessment and data extraction. Section 4 presents the results of our review. In Section 5 we discuss and interpret these results, while in Section 6 we describe potential threats to the validity of our work. Section 7 concludes the review and presents potential areas for further research and experimentation.

2 Related work

Before conducting the systematic literature review we were aware of some papers on code smell prediction and existing open access datasets of code smells. Aside from empirical studies, we also came across a few secondary studies and meta-analyses connected with the topic of code smells.

2.1 Primary studies

Fontana et al. [2] performed an empirical comparison of 16 different machine-learning algorithms on four code smells (Feature Envy, Long Method, Data Class, Large Class) and 74 software systems, with 1986 manually validated code smell samples. They found that the highest performances were obtained by J48 and Random Forest, while detection of code smells can provide high accuracy (over 96%).

Palomba et al. [14] contributed the data set containing 243 instances of five smell types from 20 open source projects manually verified by two MSc students. They also presented LANDFILL, a web-based platform aimed at promoting the collection and sharing of code smell data sets.

Palomba et al. [13] used the data set presented in [14] and proposed Historical Information for Smell deTection (HIST) approach exploiting change history information to detect instances of five different code smells (Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy). The results indicate that the precision of HIST is between 72 and 86 percent, while its recall is between 58 and 100 percent.

2.2 Secondary studies

Although a number of systematic literature reviews related with the topic of code smells exist, we found only two secondary studies reporting on their influence on bug prediction models, of which only [4] is a systematic literature review.

Cairo et al. [4] analysed 16 empirical studies to examine to what extent code smell detection influences the accuracy of bug prediction models. Their study focused primarily on the types of code smells used in the experiments, as well as the tools, techniques and resources used by researchers to find evidence of the influence of code smells on the process of fault prediction.

Gradišnik and Heričko [7] performed a review of 6 research papers published in the period from 2006 to 2014 to determine whether any subgroups of code smells influence the quality of software in a particularly harmful way. They studied the correlation between 22 distinct types of code smells and the fault-proneness of classes. Their analysis indicated weak and sometimes contradictory correlations between individual smells and the bugginess of classes. Also, they found that researchers

often focus on subgroups of smells rather than analyse their full range, their choices usually being arbitrary.

An extensive systematic review regarding ”the code smell effect” has been performed by Freitas et al. [6]. In a survey of 64 primary studies, the researchers synthesised how the concept of code smells influences the software development process. In their results, they indicated some inconsistencies between the findings of different studies. While analysing the role of people in smell detection, they found that evaluation of smells by humans has many flaws. They also created a thematic map concerning code smell themes such as ”correlation with issues of development”, ”human aspects”, ”programming” and ”detection”.

The only available systematic review on the subject written by Cairo et al. [4] was the starting point in our research. We found that an extension of the analysis is required, as the set of studies covered in the review did not include some relevant papers we found manually prior to this review (such as [15] by Palomba et al. and their studies concerning code smell intensity). The review also did not provide a synthesis enabling us to survey the already investigated smells and extract the under-researched bug prediction factors. For that reason, we decided an extended systematic literature review on the topic of the influence of code smells on bug prediction is required.

3 Research methodology

In this section we present the methodology of our systematic literature review. As suggested by Kitchenham et al. [10], this includes research questions driving the review process, description of the search strategy, selection process, as well as the approach to quality assessment and data extraction. It is important to note that throughout this paper we use the words *faults*, *defects* and *bugs* interchangeably.

3.1 Research questions

Our systematic literature review aims to answer the following research questions.

RQ1 - How does code smell detection influence the accuracy of defect-prediction?

RQ2 - Which metrics and code smells are most useful when predicting defects?

3.2 Search strategy

Our initial set of primary studies consisted of four research papers ([11], [12], [15], [17]). These papers have been found in a manual search for literature on code

smells. Next, the set of primary papers from two literature reviews ([4], [7]) was considered. These two reviews included 19 distinct papers (16 in [4] and 6 in [7] with 3 titles overlapping), but none of the four papers from our manual search. Knowing that the four papers are relevant to our research, we decided an extended primary studies search and a broader literature review must be conducted to appropriately describe the current state-of-the-art.

To extend the literature search, we decided on two additional search methods. First, we performed an automated search in the IEEE Xplore Digital Library[1]. After selecting the relevant results, we also performed forward and backward snowballing on these articles.

A search query using boolean operators was obtained for the automated search procedure. Its major expressions have been derived directly from our research questions. It consisted of a concatenation of four major terms using the operator **AND**, with possible different spellings and synonyms of each term concatenated with the use of the operator **OR**.

Source	Papers Found	Relevant	Finally selected
Manual search[2]	30	16	16
IEEE Xplore Digital Library	47	10	9
Snowballing	3	3	2
Total	80	29	27

Table 1 Search results and data sources

The search string has been defined as follows:

(*software* OR "*software project*" OR "*software projects*")

AND

("*code smell*" OR "*bad smell*" OR "*code smells*" OR "*bad smells*" OR *antipattern* OR *antipatterns* OR *anti-pattern* OR *anti-patterns* OR "*bad design*" OR "*design flaw*")

AND

(*bug* OR *bugs* OR *fault* OR *issue* OR *failure* OR *error* OR *flaw* OR *defect* OR *defects*)

AND

(*predicting* OR *prediction* OR "*prediction model*" OR *identification*)

The results of the search are presented in Table 1. The detailed depiction of the search and selection process is presented on Figure 1.

[1] https://ieeexplore.ieee.org/Xplore/home.jsp

[2] includes papers covered by [4] and [7]

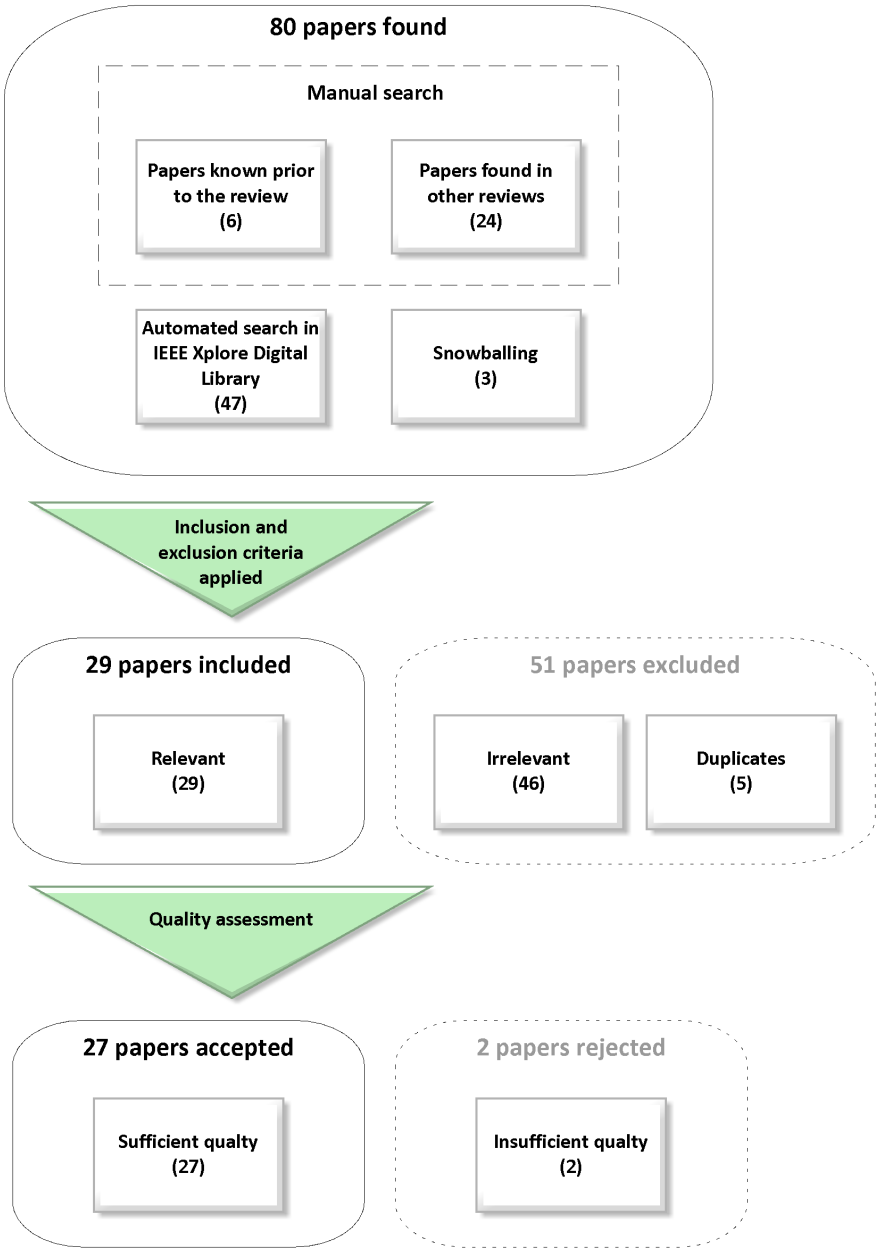

Fig. 1 The search and selection process

3.3 Selection process

First, all the results retrieved from the automated search have been preliminarily reviewed based on their titles and abstracts. While deciding on the inclusion or rejection of each article, the following criteria have been taken into consideration.

Inclusion criteria:

- Articles describing the correlation between code smells and software defects
- Articles reporting on fault prediction models based on code smell detection
- Articles which examine the influence of code smell data used as an additional predictor in other software fault prediction models
- Articles mentioning improvements to existing smell-based bug prediction techniques

Exclusion criteria:

- Articles not written in English
- Articles not related to software engineering
- Articles that focus on code smells detection techniques
- Articles focusing on the human aspect of introducing code smells

On each accepted article forward and backward snowballing has been performed. The same inclusion/exclusion criteria have been applied on the results, which lead to including 3 additional articles [9] [16] [8].

3.4 Quality assessment

To assess the quality of the selected studies, we developed a checklist of quality criteria, presented in Table 2. Its initial version was based on other literature reviews, but the checklist evolved during the initial stage of the assessment.

For each question, three possible grades were assigned - 1.0(Yes), 0.5(Partially) or 0.0(No). The grades were inserted into a spreadsheet, where they were then summed up to produce a numerical indication of each paper's quality and "suitability" for our review.

After performing the assessment it was established that only two of the considered papers are not suitable for the review, as they contained an insufficient amount of information. These two papers had the lowest quality score of 5.0 and 5.5. This way, a minimal threshold for the paper quality was established at the value 6.

3.5 Data extraction

Once the final selection of the papers was performed, we proceeded onto extracting data relevant for the review. This included reference attributes (such as title, authors,

Q1 Is the paper based on empirical evidence?
Q2 Is the research objective clearly stated?
Q3 Are the used code smells clearly defined?
Q4 Are the names of analysed software projects specified?
Q5 Does the paper state how the metrics and/or code smells data was collected?
Q6 Does the paper state the source of fault-proneness data?
Q7 Does the paper evaluate the predictive power of considered code smells in a manner allowing to draw conclusions about their influence on fault-proneness?
Q8 Is the experiment presented in the paper reproducible?
Q9 Is the conclusion clearly stated?

Table 2 Quality assessment criteria

no of pages), as well as data necessary to answer our research questions. The data extraction form attributes are presented in Table 3.

i	Title
ii	Names of the authors
iii	Year of publication
iv	Number of pages
v	Research questions
vi	Answers to research questions
vii	Types of code smells used
viii	Other metrics used
ix	Programming language
x	Analysed software projects
xi	Source of metric / code smell data
xii	Fault-proneness evaluation method
xiii	Conclusion about the influence of code smells on bug prediction
xiv	Limitations

Table 3 Data extraction form

While analysing each paper's research questions and their answers, we were primarily interested in the researchers' conclusions about the influence of code smell detection on bug prediction. In the field "Conclusion about the influence of code smells on bug prediction" we summarised this judgement to indicate whether the paper found a connection between the two or not.

We were also interested in the types of code smells used in each study, the source of data concerning faults and the statistical methods used to check whether a connection between the two exists.

4 Results

In this section, we present the results of our review. First, we briefly describe the demographics of the chosen papers. After that, we describe the results with respect to each of the research questions.

4.1 Demographics

The full list of papers selected for the review is presented at the end of this article. Most of the research papers in this area have been published in conferences. The earliest analysed paper is from 2006. The vast majority of the studies (22 out of 27) were published after 2011, which corresponds to the software engineering community's rising interest into the impact of code smells on software projects.

4.2 Code smells and bugginess

In this section we present the general conclusions of the reviewed papers concerning the influence of code smells on defect-proneness, irrespectively of the code smell type. The individual kinds of smells examined in the analysed studies are described separately in Section 4.3.

No paper provided conclusive evidence indicating that code smells are the direct source of faults. However, all of the examined papers studied one or both of the following dependencies:

- the statistical correlation between code smells and software bugs
- the influence of smell detection on the performance of bug prediction models

A. Correlation between the presence of code smells and faultiness

In general, studies aiming at exploring the correlation between code smells and bugs performed different statistical tests to check whether classes, methods and modules containing code smells contained also more defects. Table 4 presents the implications of their findings.

Zhang et al. [S27] studied the influence of 13 types of code smells on the faultiness of 18 versions of Apache Commons series software. They analysed each code smell separately and concluded that there is a positive relationship between bad design and defect-proneness. However, they pointed out that some smells point to software defects more than others.

The authors of [S22] showed in a brief study that a strong, significant relationship exists between 3 types of smells and class errors of Eclipse 2.1.

Ma et al. [12] found slight statistical agreements (Cohen's Kappa statistic in range [0.01..0.20]) between the results of fault detection and the detection of 3 out of 8 analysed code smells.

Jaafar et al. [S8] examined whether classes having static relationships with smelly classes are more defect prone. The authors conducted their studies using 11 types of code smells and found that such relationships often indicate a higher level of bug-proneness.

In their study on the influence of code smell detection on bug prediction models, the authors of [S24] performed an analysis of the co-occurrence of bugs and faults in the source code of 21 versions of two Java software projects. They found that the density of bugs (that is the number of bugs normalized with respect to file size) is generally higher in files with antipatterns as compared to files without them. They performed a Wilcoxon rank sum test, which provided statistically significant results concerning this correlation for 17 out of the 21 software versions analysed.

paper ID	positive	no link	negative
[S27]	X		
[S22]	X		
[12]	X		
[S8]	X		
[S24]	X		
[S1]	X		
[S20]		X	
[S4]	X		
[S6]		X	
[S2]	X		
[S10]	X		
[S17]	X		
[S16]	X		X
[S5]	X	X	
[S26]		X	
[S21]	X		
[S12]	X		
[S9]	X		
[S15]	X		
[S7]	X		
[S13]	X		

Table 4 Correlation between smells and bugs

The study conducted in [S1] focused on comments and whether densely commented modules contain more faults than uncommented ones. The study analysed 3 software projects (Eclipse, Apache Tomcat, Apache Ant) and concluded that there is in fact a positive correlation between the number of comments and the number of faults. This study was extended in [S2], where the authors confirmed that well-commented modules were on average 1.8 to 3 times more likely to contain bugs. However, the study results indicated that more comments do not necessarily mean more faults.

In [S20], the authors explored whether the *Clones* code smell indicates a higher level of defect-proneness. In their study, they analysed monthly snapshots of repositories of 4 software products written in C and found no such relationship. In fact, they discovered that in many cases cloned code contains on average less defects and introducing more clones into the code does not make it more buggy. They concluded that clones should not be regarded as a smell, as they are not detrimental to software quality.

In [S4], the authors attempted to determine the association between anti-patterns, maintainability and bugs. In their analysis of 34 Java projects, they found a positive correlation between the the number of faults and the number of anti-patterns (with Spearman correlation 0.55 with p below 0.001).

Hall et al. [S6] took on five less popular code smells and investigated each one's influence on defects. They obtained mixed results and no definite positive or negative correlation was found. The authors indicated that a major threat in this research area is posed by inconsistencies in the definitions of code smells between the studies.

Jaafar et al. [S10] explored the fault-proneness of classes where anti-patterns co-occur with clones. "Co-occurring" in this context did not mean that the cloned code necessarily included anti-patterns. It meant that such a class contained both some anti-patterns and some cloned code. Results of the performed study indicated that classes with such co-occurrences were at least 3 times more likely to be faulty in comparison with other classes.

Palomba et al. [S17] reported a study on the diffuseness of code smells and their influence on change- and bug-proneness based on an analysis of 30 software projects. The authors concluded that code smells have a big effect on change-proneness and a medium effect on defect-proneness. Moreover, a higher number of code smells directly indicates a higher level of fault-proneness. The paper also observed that code smells are not necessarily a direct cause of faults, but a "co-occurring phenomenon".

In [S16], the authors investigated two code smells related to size - *God Class* and *Brain Class*. They found that classes containing these two smells had more defects than those without them. However, when normalized with respect to size, these classes had less defects "per line of code". Hence, the overall conclusion about the correlation of these smells with defects is not straightforward.

In [S5], the authors reported a positive correlation between occurrences of 5 types of code smells and software defects. However, no design flaw correlated with defects more than others. The levels of correlation varied widely, depending on the analysed software project.

A multiple case study was conducted in [S26], where researchers registered the development process of four different, unspecified Java projects. The projects have been developed by six developers for a period up to four weeks. Code smells were registered before the maintenance phase and an investigation was performed to establish whether the observed faults could be caused by the smells. The authors found no causal relationship, as only around 30% of faulty files contained smells. The study also determined that introducing new smells did not increase the proportion of defects.

Saboury et al. [S21] explored the influence of 12 code smells on the "survival time" of JavaScript files. Survival time was defined as the average time until the first occurrence of a bug. The paper reported that hazard rates of the files without code smells, calculated based on their survival times, were on average 65% lower than those with smells.

In [S12], the authors explored the relationship between code smells and change- and defect-proneness, as well as whether anti-patterns are in any way related to size. In their analysis of four open source Java software projects they found that the existence of anti-patterns does indeed indicate a higher level of change and defect-proneness. However, they noted that class size itself does not influence bugginess. Their study was extended in [S9], where the process of anti-pattern mutation was analysed. By mutation the authors meant the evolution of anti-patterns over time,

especially evolution of an antipattern into a different type of antipattern. The results of this analysis indicated that anti-patterns that do not mutate are more fault-prone that the ones that are structurally altered over time.

Marinescu and Marinescu [S15] investigated the relationship between class-clients of smelly and non-smelly classes. Four types of code smells were considered. The researchers found that classes that use smelly classes are on average more defect-prone. Although such a positive correlation was found, no direct causation was proved.

In [S7], the authors explored four different techniques of technical debt detection. Apart from code smells, the other analysed techniques were modularity violations, grime buildup and automatic static analysis. The paper set out to find whether these techniques result in the same classes being classified as problematic, as well as examine the correlation between their findings and fault-proneness. The study concluded that not much overlap between all the techniques was found, but of the 9 types of code smells studied, 2 of them proved to be well-correlated with buggy classes.

In [S13], the authors investigated in a brief study the relationship between occurrences of 6 types of code smells and the error proneness of 3 versions of Eclipse. They found that while the influence widely varies between different smell types, smelliness in general does point to fault-proneness.

In summary, out of 21 papers examining the relationship between the presence of code smells and the presence of bugs, 16 reported on the existence of a positive correlation between the two. Three papers found no such link, while two papers reported inconclusive or conflicting results.

B. Influence of code smell detection on bug prediction performance

Some of the studies investigated how the information about the smelliness of classes and methods influences the accuracy of bug prediction models. Table 5 shows the general conclusions made by each study concerning their effect.

Ubayawardana and Damith Karunaratna [S25] used a bug prediction model based on 17 source code metrics as a base model. The authors introduced 4 additional, anti-pattern related metrics (relating to amount, complexity and recurrence of anti-patterns), as well as an intensity index. They compared the performance of the model with and without the smell-related metrics using three different classifiers trained on 13 different software projects. Their results showed that bug prediction based solely on source code metrics is not reliable. However, including the anti-pattern metrics significantly enhanced the performance of the models. In some cases, with the use of the Random Forest classifier, the authors reported achieving 100% accuracy of bug prediction on test data.

The study conducted in [12] found only a small agreement between code smell detection and the results of fault prediction using a multivariable logistic regression model based on cohesion metrics. However, the authors noted that using code smell detection results can improve the recall of bug prediction, as smelliness often points to faultiness. Using combined data of all the considered code smells, the fault prediction results were improved by 9 to 16%.

In [S11], the authors performed an analysis of how sampling and resampling techniques influence the performance of bug prediction models based on source code metrics and code smells. Their study concluded that code smells themselves are not a better predictor than metrics.

paper ID	positive	neutral	negative
[S25]	X		
[12]	X		
[S11]			X
[S24]	X		
[S18]	X		
[S19]	X		
[S4]	X		
[S3]	X		
[S23]	X		

Table 5 Influence on bug prediction accuracy

Taba et al. [S24] used as a base model a logistic regression bug prediction model based on source code metrics that proved to be accurate in previous studies. They developed four smell-related metrics of their own and added them to the model, one by one, to test its performance. The best results were achieved with the use of their "Anti-pattern Recurrence" metric, which proved to improve the accuracy of the baseline model by 12.5% on average.

In [S18], the authors introduced a code smell intensity index as a bug predictor. For their baseline model they used 20 code metrics proposed by other researchers. Their study covered how the introduction of smell intensity affects the performance of the base model. They found that code smell intensity always positively influences the accuracy of bug prediction. Furthermore, their analysis indicated that smell intensity provides additional information that a simple indicator of smell presence could not give. The authors continued this study in [S19], where they verified the influence of their intensity index against models based on product and process metrics, as well as metrics related to the history of code smells in files. Their results indicated that smell intensity is an important predictor and its use always increases the accuracy of bug prediction.

In [S4], while analysing the impact of code smells on software maintainability, the authors observed that smells also show significant predictive power. They noted that the number of anti-patterns appearing in a class is only a slightly worse predictor than the intricate metric model they have studied as their main subject.

Aman et al. [S3] investigated whether *Lines Of Comments* (LCM) can serve as a useful metric for predicting defects in software. They used this metric to categorize methods as more-commented or less-commented. They found that more-commented methods are on average 1.6-2.8 as likely to contain bugs. Furthermore, they used existing fault data to train logistic regression models and found that LCM is a useful bug predictor, alongside metrics related to the method's size and complexity.

In [S23], the authors built defect prediction models with the use of code smell and churn metrics. They investigated and compared the impact of those metrics on bug prediction performance. They reported that code smell metrics significantly improved the performance of the models. The machine learning algorithms they used, Logistic Regression and Naive Bayes, also performed better with code smell metrics than churn metrics.

To sum up, eight out of nine studies covering the effect of code smell-related information on the accuracy of bug prediction reported positive results. The one

remaining study described the code smells' defect prediction ability as no greater than the one of source code metrics.

4.3 The impact of individual smells

A. Landscape of studied code smells

Table 6 presents smells and smell-related metrics studied in the analysed research papers. The smells are sorted in a descending order starting from most frequently studied.

The most popular smells are those connected with large size and high complexity of classes and methods, such as *Blob*, *God Method*, *Complex Class* or *Long Parameter List*. Other frequently studied smells are related to improper inheritance (*Refused Parent Bequest*), improper handling of class attributes (*Class Data Should Be Private*) and bad encapsulation (*Feature Envy*, *Lazy Class*, *Shotgun Surgery*).

Among the less studied anti-patterns there are such smells as *Inappropriate Intimacy*, *Base Class Should Be Abstract*, *Temporary Field*, *Middle Man*, *Switch Statements* or *Data Clumps*. Also, the set of anti-patterns used in [S21] contains many unique smells, as the authors used some JavaScript-specific smells, taken from popular JavaScript coding guides.

B. Usefulness and popularity of smells

While some papers treated code smells collectively, using as a predictor an indication of whether any type of smell was present in a given class or method, most studies evaluated them separately. Studies of the second type were often able to draw conclusions about which types of smells proved to be most useful in the task of bug prediction. Table 7 summarises those conclusions, including cases where researchers found little or close to no influence of smells on the level of software faultiness.

The information from Tables 6 and 7 has been aggregated into Table 8. This table combines information concerning all the code smells and metrics used as bug predictors in the analysed studies and allows to evaluate their popularity as well as potential usefulness.

The code smell that was most frequently indicated as highly correlated with bugs was *God Class*. Researchers reported high usefulness of this smell in 8 out of 18 analysed studies. Together with high reported usefulness of *God Method* and *Brain Class* this indicates that big sizes and high complexities of classes and methods are both popular and effective in the task of bug prediction. One notable exception from this trend was presented in [S16]. In this study, authors found that *God Classes* and *God Methods* do have on average more faults than other classes. However, they also found that when the number of bugs is normalized with respect to the size of classes or methods, they are in fact less buggy. Their study pointed out that most studies analysing *God Class* and *God Method* take into consideration absolute, not relative values of bug-proneness of those classes.

smell (smell metric)	paper IDs
Blob / God Class	[S27] [S22] [12] [S11] [S8] [S24] [S18] [S19] [S4] [S10] [S17] [S16] [S26] [S12] [S9] [S15] [S7] [S13]
Long / God / Brain Method	[S27] [S22] [S11] [S8] [S24] [S18] [S19] [S4] [S10] [S17] [S16] [S5] [S26] [S21] [S12] [S9] [S13]
Brain / Complex Class	[S27] [12] [S8] [S24] [S10] [S17] [S16] [S21] [S12] [S9] [S15] [S7]
Refused Parent Bequest	[S27] [12] [S8] [S24] [S4] [S10] [S17] [S26] [S12] [S9] [S7] [S13]
Long Parameter List	[S27] [12] [S8] [S24] [S4] [S10] [S17] [S21] [S12] [S9]
Message Chain	[S27] [S8] [S24] [S18] [S19] [S6] [S10] [S17] [S12] [S9]
Class Data Should Be Private	[S27] [12] [S8] [S24] [S10] [S17] [S12] [S9]
Feature Envy	[S11] [S4] [S17] [S5] [S26] [S15] [S7] [S13]
Lazy Class	[S27] [12] [S24] [S4] [S10] [S17] [S12] [S9]
Shotgun Surgery	[S22] [S18] [S19] [S4] [S5] [S26] [S7] [S13]
Speculative Generality	[S27] [S8] [S24] [S6] [S10] [S17] [S12] [S9]
Anti Singleton	[S27] [12] [S8] [S24] [S10] [S12] [S9]
Data class	[S18] [S19] [S10] [S26] [S15] [S7] [S13]
Swiss Army Knife	[S27] [12] [S8] [S24] [S10] [S12] [S9]
Large Class	[S27] [S24] [S4] [S10] [S12] [S9]
Spaghetti Code	[S27] [S8] [S24] [S10] [S17] [S12]
Dispersed Coupling	[S18] [S19] [S5] [S7]
Clones / Duplicated Code	[S20] [S10] [S26]
Comments (LOC)	[S1] [S2] [S3]
Data clumps	[S6] [S26]
Intensive coupling	[S5] [S7]
Middle Man	[S6] [S17]
Switch Statement	[S6] [S21]
Variable Re-assign	[S26] [S21]
Base Class Should Be Abstract	[S10]
Empty catch block, dummy handler, unprotected main programs, nested try statement, careless cleanup, exceptions thrown from finally block	[S11]
Inappropriate Intimacy	[S17]
Lengthy Lines, Chained Methods, Nested Callbacks, Assignment in Conditional Statements, Extra Bind, This Assign, Depth	[S21]
Misplace Class	[S26]
Temporary field	[S4]
Tradition breaker	[S7]
unknown	[S23]
Use interface instead of implementation, Interface Segregation Principle Violation	[S26]

Table 6 Smells studied in the analysed papers

Other popular smells concerned the inter-relations of classes and the communication between them. *Message Chains* was concerned a good indicator of faultiness

factor	papers reporting high usefulness	papers reporting low usefulness
Blob / God Class	[S27] [12] [S8] [S18] [S19] [S17] [S7] [S13]	[S16]
Long / God / Brain Method	[S27] [S22] [S18] [S19] [S12]	
Message Chain	[S18] [S19] [S12] [S9]	
Dispersed Coupling	[S18] [S19] [S7]	
Comments (LCM)	[S1] [S2] [S3]	
Brain / Complex Class	[S27] [S8] [S12]	[S16]
Anit-pattern / smell intensity	[S25] [S18] [S19]	
Avg no of anti-patterns	[S25] [S19] [S4]	
Refused Parent Bequest	[12] [S17] [S9]	[S22] [S13]
Shotgun Surgery	[S22] [S19]	
Anti-pat. recurrence length	[S25] [S24]	
LinesOfCode (size)	[S27] [S3]	[S6]
Data class	[S18] [S19]	[S22] [S13]
Anti Singleton	[S12]	
Assignment in Cond. Statements	[S21]	
Clones / Duplicated Code	[S10]	
Inappropriate Intimacy	[S17]	
Large Class	[S22]	
Long Parameter List	[S27]	
Variable Re-assign	[S21]	
CallDependency	[S27]	
Cyclomatic Complexity	[S3]	
Anti-pat. complexity	[S25]	
Anti-pat. cumulative pairwise diff.	[S25]	
Anti-pat. indicator (present/absent)	[S10]	
Lazy Class	[S12]	[12]
Swiss Army Knife	[S8]	[12]
Feature Envy	[S17]	[S22] [S13]
CDSBP		
Spaghetti Code		
Middle Man		[S6]
Speculative Generality		[S6]
Switch Statement		[S6]

Table 7 Papers reporting on smell usefulness

in 4 papers (out of 10 covering this anti-pattern). *Dispersed Coupling* and *Refused Parent Bequest* were both described as useful by 3 out of 12 papers that concerned these smells. However, Shatnawi and Li [S22] regarded *Refused Parent Bequest* as too infrequent to analyse, while Li and Shatnawi [S13] concluded that its influence on error rates is very small.

Aman et al. [1] analysed the influence of *Comments* in studies [S1], [S2] and [S3]. All those studies indicated a high correlation between well-commented and buggy modules (1.6-3 times higher than in other classes). The authors point out that this makes *Comments* a useful indicator of bugginess.

Extensive studies conducted by Palomba et al. [15] in [S18] and [S19] proved that code *smell intensity* is a useful bug predictor. The inclusion of *smell intensity* as a predictor improved the performance of bug prediction models based on process,

product and anti-pattern metrics. The usefulness of *smell intensity* as a predictor was also confirmed by the empirical study performed in [S25].

A similar metric, *Average number of anti-patterns* was studied in [S25], [S19] and [S4]. In all three studies the authors concluded that information about how smelly a class is, in the sense of how many code smells it contains, is also useful in bug prediction.

Some popular smells proved to have a much smaller effect on bug-proneness. Smells such as *Long Parameter List*, *Anti-singleton* and *Large Class* were deemed useful only by one paper, despite being covered subsequently by 10, 7 and 6 papers each.

The *Data Class* smell was considered a very useful predictor by [S18] and [S19]. However, Shatnawi and Li [S22][S13] did not find a significant correlation between this smell and bug-proneness.

In their research, Hall et al. [S6] showed that the *Switch Statement* smell has no effect on bug-proneness of any of the studied software projects. *Middle Man* and *Speculative Generality* provided mixed results that varied depending on the project.

Conflicting results were reported regarding smells *Lazy Class*, *Swiss Army Knife* and *Feature Envy*. Despite these smells being studied relatively often, only singular papers were able to reach any clear (although often contradictory) conclusions about their predictive power. Other quite popular smells, such as *Class Data Should Be Private* and *Spaghetti Code* (described in 6 and 8 articles respectively) were not singled-out even once, their influence usually being present but minimal.

5 Discussion

In this section we summarise the findings of our review and address the proposed research questions.

RQ1. *How does code smell detection influence the accuracy of bug-prediction?*

The great majority of analysed research papers found a positive correlation between code smells and software bugs. Although the extent of how strong and significant this relationship is varied depending on the smell types and software projects analysed, a general conclusion can be drawn that code smells are a potentially good predictor of software defects. It needs to be noted, however, that a substantial number of studies found no direct link between the studied types of smells and the bugginess of analysed software.

Many studies attributed these varying levels of code smell harmfulness in different projects to distinct company policies and software engineering practices employed in the process of software development. With different development practices, the same code smells may induce detrimental effects on software quality or have no influence at all.

Another cause of singular contradictory results might be the large diversity of code smell definitions, as well as different fault and smell data gathering meth-

type	factor	# +	# –	# papers
s	Blob / God Class	8	1	18
s	Long / God / Brain Method	5	0	17
s	Message Chain	4	0	10
s	Brain / Complex Class	3	1	12
s	Dispersed Coupling	3	0	4
s	Comments (LCM)	3	0	3
m	Anti-pattern / smell intensity	3	0	3
m	Avg no of anti-patterns	3	0	3
s	Refused Parent Bequest	3	2	12
s	Shotgun Surgery	2	0	8
m	Anti-pattern recurrence length	2	0	2
m	LinesOfCode (size)	2	1	3
s	Data class	2	2	7
s	Long Parameter List	1	0	10
s	Anti Singleton	1	0	7
s	Large Class	1	0	6
s	Clones / Duplicated Code	1	0	3
s	Variable Re-assign	1	0	2
m	Anti-pattern complexity	1	0	2
m	Anti-patten indicator (present/absent)	1	0	2
s	Assignment in Conditional Statements	1	0	1
s	Inappropriate Intimacy	1	0	1
m	Call Dependency	1	0	1
m	Cyclomatic Complexity	1	0	1
m	Anti-pattern cumulative pairwise differences	1	0	1
s	Lazy Class	1	1	8
s	Swiss Army Knife	1	1	7
s	Feature Envy	1	2	8
s	CDSBP	0	0	8
s	Spaghetti Code	0	0	6
s	Speculative Generality	0	1	8
s	Middle Man	0	1	2
s	Switch Statement	0	1	2

Table 8 Number of papers reporting on individual smells - summary
Legend:
s: code smell
m: metric;
#+: number of papers reporting high usefulness
 # –: number of papers reporting low usefulness
#papers: total number of papers reporting

ods/tools. Lack of unified methodology makes comparing the results of different studies difficult.

Synthesising the results of studies focusing on the predictive power of code smells in bug prediction models allowed to reach a clearer conclusion. Nearly all analysed papers concluded that the inclusion of code smell information improves the accuracy of bug prediction in virtually any previously devised prediction model. The smallest effect was reported by a study that described the influence of code smells as comparable to that of source code metrics. This fact reaffirms the usefulness of code smells as bug predictors.

RQ2. *Which metrics and code smells are most useful when predicting defects?*

Code smells relating to extensively big classes and methods of high complexity, such as *God Class* and *God Method* were not only the most frequently studied, but were also the smells that proved to be very well correlated with software bugs in the largest number of empirical studies. The interest into those smells, as well as reports of their effectiveness correspond with the common belief of the software development community that writing large code modules of high complexity is a bad practice.

Message Chains and *Dispersed Coupling* smells were relatively often singled out as exceptionally good bug predictors. Out of 9 studies reporting a positive influence of a set of code smells including *Message Chains*, 5 of them underlined this anti-pattern's exceptionally high positive correlation with bugs. In the case of *Dispersed Coupling*, this ratio was 3 out of 4.

Also *Comments* proved to be valuable bug predictors, reaffirming the common clean code practice that source code should be self-explanatory. Although the number of comments itself was not directly related to the number of bugs, the in-module comment presence indicator proved to be a valuable predictor, signalling pieces of code that are overly complicated, unclear and potentially faulty.

Good bug prediction results were also achieved by anti-pattern metrics. Especially *code smell intensity* deserves to be mentioned, as the extensive studies investigating its bug prediction power reported it to increase the accuracy of all common types of previously tested bug prediction models, reaching exceptionally good results.

Also *Average number of anti-patterns* and *Anti-pattern recurrence length* smell-related metrics proved to be successfully used as bug predictors alongside other factors.

6 Threats to validity

The primary threat to validity in any systematic literature review is related to the completeness of the set of analysed studies. We put particular attention to developing an elaborate search term in order to obtain a comprehensive set of potentially relevant papers. However, the set of analysed papers could be biased due to limiting our automated search to the IEEE Xplore Digital Library. We chose this database based on our experience with the results of a few pilot searches. IEEE results included all the papers we were previously familiar with, as well as relevant papers we found in other databases such as ACM and ProQuest. IEEE also performed well with long search terms and consistently provided concise and relevant results. Any singular studies related with our research questions that we could have left out due to the above limitations should not invalidate the overall conclusions of our review, taking into consideration its qualitative nature.

Another factor that could influence the validity of this review is positive publication bias. Of all the relevant papers we have found, only a small minority reports negative results which undermine the usefulness of code smells as bug predictors. In

our review we assumed that this trend corresponds with the reality and the majority of studies reporting positive results can be treated as an indicator of an existing correlation between smells and bugs.

It is also worth to note a few recurring validity threats of the analysed papers themselves. Although they are not related to the methodology of our review, they can be in some way related to its conclusions. Most papers dealt only with Java projects. Single studies analysing C, JavaScript or PHP projects took into consideration code smell types that differed from the ones used in Java. An insufficient amount of research has been conducted so far to determine whether the smells-bugs dependency is similar in projects written in other programming languages and draw conclusions about its generality.

Also, many studies did not conduct any tests to verify the accuracy of smell detection. The authors generally leaned on other research papers proving the accuracy of chosen smell detection tools or devised their own, metric-based smell finders. All the analysed studies assumed that the results obtained by their smell detection techniques were accurate.

7 Conclusions

This systematic literature review aimed at describing the state-of-the-art of the research on the use of code smells in bug prediction, as well as providing an analysis of the usefulness of different code smell-related factors. To achieve this aim, we conducted an analysis of 27 research papers, paying particular attention to the information concerning:

- the general contribution of smell detection to bug prediction models,
- the variety of used smells and metrics and their usefulness,
- the areas of related research that remain insufficiently explored.

The results of our work show that code smells are indeed a good indicator of bugs. However, their usefulness differs depending on the type of anti-pattern and metric analysed, as well as the software project in which they are tested. Among all the analysed types of smells, *God Class*, *God Method* and *Message Chains* stand out. Very good results were also obtained with the use of smell-related metrics, especially *Code smell intensity*.

Our survey also disclosed a large group of existing code smell types that remain very scarcely researched. Smells such as *Inappropriate Intimacy*, *Variable Re-assign* and *Clones* showed promising bug-predicting properties in individual studies, but this effect requires further validation.

Middle Man and *Speculative Generality* were only analysed in two empirical studies, providing inconclusive results. Their usefulness in bug prediction is also a potential field for further research.

Good results were achieved with the use of metrics describing the structure and intensity of code smells found in software systems, as well as the history of their

introduction and evolution. This type of data, along with information regarding the coupling and co-occurences of different types of code smells might constitute a good field for experiments and can provide an opportunity to develop a new, useful code smell-related metric to use as a bug predictor.

Papers analysed in the systematic literature review

1. Aman, H.: An empirical analysis on fault-proneness of well-commented modules. In: Proceedings - 2012 4th International Workshop on Empirical Software Engineering in Practice, IWESEP 2012, pp. 3–9 (2012). DOI 10.1109/IWESEP.2012.12
2. Aman, H., Amasaki, S., Sasaki, T., Kawahara, M.: Empirical analysis of fault-proneness in methods by focusing on their comment lines. In: Proceedings - Asia-Pacific Software Engineering Conference, APSEC (2014). DOI 10.1109/APSEC.2014.93
3. 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). DOI 10.1587/transinf.2015EDP7107
4. Bán, D., Ferenc, R.: Recognizing antipatterns and analyzing their effects on software maintainability, vol. 8583 LNCS (2014). DOI 10.1007/978-3-319-09156-3_25
5. D'Ambros, M., Bacchelli, A., Lanza, M.: On the impact of design flaws on software defects. Proceedings - International Conference on Quality Software (1), 23–31 (2010). DOI 10.1109/QSIC.2010.58
6. 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 (2014). DOI 10.1145/2629648
7. Izurieta, C., Seaman, C., Cai, Y., Shull, F., Zazworka, N., Wong, S., Vetro', A.: Comparing four approaches for technical debt identification. Software Quality Journal **22**(3), 403–426 (2013). DOI 10.1007/s11219-013-9200-8
8. Jaafar, F., Guéhéneuc, Y., Hamel, S., Khomh, F.: Mining the relationship between anti-patterns dependencies and fault-proneness. In: 2013 20th Working Conference on Reverse Engineering (WCRE), pp. 351–360 (2013). DOI 10.1109/WCRE.2013.6671310
9. Jaafar, F., Khomh, F., Gueheneuc, Y.G., Zulkernine, M.: Anti-pattern mutations and fault-proneness. In: Proceedings - International Conference on Quality Software, pp. 246–255 (2014). DOI 10.1109/QSIC.2014.45
10. Jaafar, F., Lozano, A., Gueheneuc, Y.G., Mens, K.: On the analysis of co-occurrence of anti-patterns and clones. In: Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017 (2017). DOI 10.1109/QRS.2017.38
11. Kaur, K., Kaur, P.: Evaluation of sampling techniques in software fault prediction using metrics and code smells. In: 2017 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2017, vol. 2017-Janua, pp. 1377–1386. IEEE (2017). DOI 10.1109/ICACCI.2017.8126033
12. Khomh, F., Penta, M.D., Guéhéneuc, Y.G., Antoniol, G.: An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empirical Software Engineering **17**(3), 243–275 (2012). DOI 10.1007/s10664-011-9171-y
13. 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 **80**(7), 1120–1128 (2007). DOI 10.1016/j.jss.2006.10.018
14. Ma, W., Chen, L., Zhou, Y., Xu, B.: Do we have a chance to fix bugs when refactoring code smells? Proceedings - 2016 International Conference on Software Analysis, Testing and Evolution, SATE 2016 pp. 24–29 (2016). DOI 10.1109/SATE.2016.11
15. Marinescu, R., Marinescu, C.: Are the clients of flawed classes (also) defect prone? Proceedings - 11th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2011 pp. 65–74 (2011). DOI 10.1109/SCAM.2011.9

16. 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: IEEE International Conference on Software Maintenance, ICSM (2010). DOI 10.1109/ICSM.2010.5609564
17. Palomba, F., Bavota, G., Penta, M.D., Fasano, F., Oliveto, R., Lucia, A.D.: On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering (2018). DOI 10.1007/s10664-017-9535-z
18. Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R.: Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016 pp. 244–255 (2017). DOI 10.1109/ICSME.2016.27
19. Palomba, F., Zanoni, M., Fontana, F.A., Lucia, A.D., Oliveto, R.: Toward a Smell-Aware Bug Prediction Model. IEEE Transactions on Software Engineering **45**(2), 194–218 (2019). DOI 10.1109/TSE.2017.2770122
20. Rahman, F., Bird, C., Devanbu, P.: Clones: What is that smell? Empirical Software Engineering **17**(4-5), 503–530 (2012). DOI 10.1007/s10664-011-9195-3
21. Saboury, A., Musavi, P., Khomh, F., Antoniol, G.: An empirical study of code smells in JavaScript projects. In: SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering, pp. 294–305 (2017). DOI 10.1109/SANER.2017.7884630
22. Shatnawi, R., Li, W.: An investigation of bad smells in object-oriented design. In: Proceedings - Third International Conference onInformation Technology: New Generations, ITNG 2006, vol. 2006, pp. 161–163 (2006). DOI 10.1109/ITNG.2006.31
23. Soltanifar, B., Akbarinasaji, S., Caglayan, B., Bener, A.B., Filiz, A., Kramer, B.M.: Software Analytics in Practice: A Defect Prediction Model Using Code Smells. Proceedings of the 20th International Database Engineering & Applications Symposium on - IDEAS '16 pp. 148–155 (2016). DOI 10.1145/2938503.2938553
24. Taba, S.E.S., Khomh, F., Zou, Y., Hassan, A.E., Nagappan, M.: Predicting bugs using antipatterns. IEEE International Conference on Software Maintenance, ICSM pp. 270–279 (2013). DOI 10.1109/ICSM.2013.38
25. Ubayawardana, G.M., Damith Karunaratna, D.: Bug Prediction Model using Code Smells. 2018 18th International Conference on Advances in ICT for Emerging Regions (ICTer) pp. 70–77 (2019). DOI 10.1109/icter.2018.8615550
26. Yamashita, A., Moonen, L.: To what extent can maintenance problems be predicted by code smell detection? -An empirical study. Information and Software Technology (2013). DOI 10.1016/j.infsof.2013.08.002
27. Zhang, X., Zhou, Y., Zhu, C.: An empirical study of the impact of bad designs on defect proneness. In: Proceedings - 2017 Annual Conference on Software Analysis, Testing and Evolution, SATE 2017, vol. 2017-Janua, pp. 1–9 (2017). DOI 10.1109/SATE.2017.9

References

1. Aman, H., Amasaki, S., Sasaki, T., Kawahara, M.: Empirical analysis of fault-proneness in methods by focusing on their comment lines. In: Proceedings - Asia-Pacific Software Engineering Conference, APSEC (2014). DOI 10.1109/APSEC.2014.93
2. Arcelli Fontana, F., Mäntylä, M.V., Zanoni, M., Marino, A.: Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering **21**(3), 1143–1191 (2016)
3. Azeem, M.I., Palomba, F., Shi, L., Wang, Q.: Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Information and Software Technology **108**(4), 115–138 (2019). DOI 10.1016/j.infsof.2018.12.009. URL `https://doi.org/10.1016/j.infsof.2018.12.009`

4. Cairo, A.S., Carneiro, G.d.F., Monteiro, M.P.: The impact of code smells on software bugs: A systematic literature review. Information (Switzerland) **9**(11), 1–22 (2018). DOI 10.3390/info9110273
5. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring Improving the Design of Existing Code - Fowler-Beck-Brant-Opdyke-Roberts. Xtemp01 (1999)
6. Freitas, M.F., Santos, J.A.M., do Nascimento, R.S., de Mendonça, M.G., Rocha-Junior, J.B., Prates, L.C.L.: A systematic review on the code smell effect. Journal of Systems and Software **144**(October 2016), 450–477 (2018). DOI 10.1016/j.jss.2018.07.035
7. Gradišnik, M., Heričko, M.: Impact of code smells on the rate of defects in software: A literature review. CEUR Workshop Proceedings **2217**, 27–30 (2018)
8. Jaafar, F., Khomh, F., Gueheneuc, Y.G., Zulkernine, M.: Anti-pattern mutations and fault-proneness. In: Proceedings - International Conference on Quality Software, pp. 246–255 (2014). DOI 10.1109/QSIC.2014.45
9. Khomh, F., Penta, M.D., Guéhéneuc, Y.G., Antoniol, G.: An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empirical Software Engineering **17**(3), 243–275 (2012). DOI 10.1007/s10664-011-9171-y
10. Kitchenham, B.A., Budgen, D., Brereton, P.: Evidence-Based Software Engineering and Systematic Reviews. Chapman & Hall/CRC (2015)
11. Le, D., Medvidovic, N.: Architectural-based speculative analysis to predict bugs in a software system. In: Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16, pp. 807–810. ACM, New York, NY, USA (2016). DOI 10.1145/2889160.2889260. URL http://doi.acm.org/10.1145/2889160.2889260
12. Ma, W., Chen, L., Zhou, Y., Xu, B.: Do we have a chance to fix bugs when refactoring code smells? Proceedings - 2016 International Conference on Software Analysis, Testing and Evolution, SATE 2016 pp. 24–29 (2016). DOI 10.1109/SATE.2016.11
13. Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Poshyvanyk, D., Lucia, A.D.: Mining version histories for detecting code smells. IEEE Transactions on Software Engineering **41**(5), 462–489 (2015)
14. Palomba, F., Nucci, D.D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D., De Lucia, A.: Landfill: An open dataset of code smells with public evaluation. In: Proceedings of the 12th Working Conference on Mining Software Repositories, MSR '15, pp. 482–485. IEEE Press, Piscataway, NJ, USA (2015)
15. Palomba, F., Zanoni, M., Fontana, F.A., De Lucia, A., Oliveto, R.: Smells like teen spirit: Improving bug prediction performance using the intensity of code smells. Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016 pp. 244–255 (2017). DOI 10.1109/ICSME.2016.27
16. Singh, S., Kahlon, K.S.: Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. ACM SIGSOFT Software Engineering Notes **36**(5), 1 (2011). DOI 10.1145/2020976.2020994
17. Soltanifar, B., Akbarinasaji, S., Caglayan, B., Bener, A.B., Filiz, A., Kramer, B.M.: Software Analytics in Practice: A Defect Prediction Model Using Code Smells. Proceedings of the 20th International Database Engineering & Applications Symposium on - IDEAS '16 pp. 148–155 (2016). DOI 10.1145/2938503.2938553