Information and Software Technology 115 (2019) 79–92

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

The usefulness of software metric thresholds for detection of bad smells and fault prediction

Mariza A.S. Bigonha[a,*], Kecia Ferreira[b], Priscila Souza[a], Bruno Sousa[a], Marcela Januário[b], Daniele Lima[b]

[a] Computer Science Department, UFMG, Brazil
[b] Department of Computing, CEFET-MG, Brazil

ARTICLE INFO

Keywords:
Software metrics
Software quality
Thresholds
Detection strategies
Bad smell
Fault prediction

ABSTRACT

Context: Software metrics may be an effective tool to assess the quality of software, but to guide their use it is important to define their thresholds. Bad smells and fault also impact the quality of software. Extracting metrics from software systems is relatively low cost since there are tools widely used for this purpose, which makes feasible applying software metrics to identify bad smells and to predict faults.

Objective: To inspect whether thresholds of object-oriented metrics may be used to aid bad smells detection and fault predictions.

Method: To direct this research, we have defined three research questions (RQ), two related to identification of bad smells, and one for identifying fault in software systems. To answer these RQs, we have proposed detection strategies for the bad smells: Large Class, Long Method, Data Class, Feature Envy, and Refused Bequest, based on metrics and their thresholds. To assess the quality of the derived thresholds, we have made two studies. The first one was conducted to evaluate their efficacy on detecting these bad smells on 12 systems. A second study was conducted to investigate for each of the class level software metrics: DIT, LCOM, NOF, NOM, NORM, NSC, NSF, NSM, SIX, and WMC, if the ranges of values determined by thresholds are useful to identify fault in software systems.

Results: Both studies confirm that metric thresholds may support the prediction of faults in software and are significantly and effective in the detection of bad smells.

Conclusion: The results of this work suggest practical applications of metric thresholds to identify bad smells and predict faults and hence, support software quality assurance activities.Their use may help developers to focus their efforts on classes that tend to fail, thereby minimizing the occurrence of future problems.

1. Introduction

Software metrics provide quantitative evaluation of software quality, allowing engineers to change the course of the process to promote quality improvement of the final product. Despite the importance of metrics in managing quality of object-oriented software (OOS), they are not yet effectively used [14,41,49]. One possible reason is that to perform effective software measurement and increase the use of metrics it is essential to associate them with thresholds [3,14,15].

Definition of methods for deriving thresholds for metrics is the focus of several studies [3,14,15,25,31,32,38,43,45,53]. Although there are several metrics proposed and a reasonable amount of defined methods to derive thresholds, there are few metrics with threshold, which makes it difficult and even impossible to manage the quality of software by quantitative means.

Additionally, although thresholds for software metrics have been evaluated, such evaluations are not widespread. It is important to consider whether they are useful to identify bad smells and to predict faults, which are attributes that directly impact the quality of software.

Bad smells are features found in code structures that indicate they are troublesome and need to be restructured. Bad smells are not the direct cause of faults but may indirectly influence the insertion of faults responsible for future faults [18]. Marinescu [30] is possibly the first researcher to define strategies for detecting bad smells using metrics. Lanza and Marinescu [26] also define a list of bad smells, some new and some modified from Fowler's work [18], and use detection strategies to

* Corresponding author.
E-mail addresses: mariza@dcc.ufmg.br (M.A.S. Bigonha), kecia@decom.cefetmg.br (K. Ferreira), priscilinhapsouza@gmail.com (P. Souza), bruno.luan.sousa@dcc.ufmg.br (B. Sousa), marcelajanuario92@hotmail.com (M. Januário), danieleddelima@gmail.com (D. Lima).

https://doi.org/10.1016/j.infsof.2019.08.005
Received 5 March 2018; Received in revised form 22 May 2019; Accepted 5 August 2019
Available online 6 August 2019

identify them. Other works propose detection of bad smell from UML and class diagrams [37] and by means of metrics of interest [39]. Zhang et al. [58] use the CPD (Copy/Paste Detector) function of the PDM open source tool[1] and a pattern-based Code Smells detection tool to identify bad smells.

Many studies were carried out to investigate the relationship between fault and aspects such as metrics, thresholds, [2,27,31,32], and bad smells [8,20]. Fault prediction techniques involve analysis of the source code of a version of a system [5] or historical data from repositories [35], fault tracking systems [1,4,22].

The purpose of such techniques is to predict the occurrence of faults found in each component of the system, for example: package, class, method or any other code unit [11].

Extracting metrics from software systems is relatively low cost, there are tools available with this purpose: CodePro Analytix,[2] Metrics[3] and Sonar Qube.[4] Thus, the use of metrics to predict faults may be more feasible than the application of complex techniques like Machine Learning and Source Code Analysis. It is important to investigate the efficacy of metric-threshold values in this context.

The aim of this research is to perform a comprehensive empirical study to evaluate the applicability of software metric thresholds in the identification of four bad smells defined by Fowler [18] (Data Class, Large Class, Long Method, and Refused Bequest) and in fault prediction. For this purpose, we use the OOS metric thresholds of Filó et al. [15]. We have chosen these thresholds because they belong to one of the largest catalogue for object-oriented metrics thresholds, considering 18 metrics. The catalogue of Filó's et al. is an extension of the thresholds proposed by Ferreira et al. [14]. In their work, Ferreira et al. show that the proposed thresholds are applicable regardless the size, application domain, and type of the systems (tool, framework, and library). This is another reason we chose to use the Filó's catalogue in our work. Moreover, this catalogue was previously used to evaluate the internal quality of a proprietary software system and the results suggest that the proposed thresholds provided a proper assessment of the system.

To direct this research we have defined the following research questions (RQ):

RQ1 *Do thresholds of object-oriented software metrics help to identify bad smells?*

To answer RQ1 we propose, in Section 2, detection strategies of five bad smells based on metric-threshold values [48], since previous studies reveal the absence of research focusing on the use of thresholds for bad smells identification [26,30,39]. This generic question is divided into two more specific ones.

RQ1.1 *What is the effectiveness of detecting bad smells using strategies based on the thresholds and taking into account the results generated by bad smells detection tools?*

The objective of the experiment to answer RQ1.1 is to place the results of our detection strategies in perspective with the results of two tools that automatically identify bad smells. Sections 2.1.4–2.1.6, respectively, presents the experiment and the analysis made to answer RQ1.1.

RQ1.2 *Do thresholds effectively support the detection of bad smells against reference lists generated by specialists with object-oriented knowledge and bad smells?*

We have used tools to automatically detect bad smells in the analysis referring to RQ1.1. However, the effectiveness of these tools is not known and there are differences between the results generated by them. To better evaluate the detection strategies proposed, we have invited three specialists with good knowledge in OOP and bad smells to compose and validate reference list. Section 3 describes the experiment conducted to answer RQ1.2.

RQ2. *Do thresholds of object-oriented software metrics help to predict software fault?*

With this RQ2, we intend to verify the usefulness of threshold values, Regular/Occasional and Bad/Rare [15], in predicting software faults. Section 4 presents the experiment and the analysis conducted to answer RQ2.

2. Threshold and bad smells

To conduct our investigation, we have used a threshold catalog of 18 software metrics [15] derived from 100 software systems [50] to define detection strategies for the bad smells: Large Class, Long Method, Data Class, Feature Envy and Refused Bequest [48]. We examine the effectiveness of the thresholds, analyzing the bad smells detected using these strategies in 12 software extracted from Qualitas.class corpus [50]. The results have been compared with the results obtained by the tools JSpIRIT [52] and JDeodorant [51], used to identify bad smells automatically.

2.1. Strategies for detecting bad smells

Detection strategy may be defined as an expression quantification of a rule used to verify if fragments of the source code complies with this rule. The components used for definition of bad smells are metric filtering mechanisms and mechanisms of composition. Filtering is used to reduce the initial set of data, capturing fragments of code that match with the filters. Filters allow to set value limits for certain aspects. Composition allows different filters, based on different software metrics, by means of logical connectives, thus enabling the mixture of metrics for relevant information [30]. To define data filter one must define values for the lower and upper limits of the filtered subset. Filters may be statistical, based on absolute or relative thresholds [26].

To define the filtering and composition mechanisms and propose the strategies to detect bad smells, the software metrics were chosen according to their availability in the catalog illustrated in Table 1. The bad smells chosen are those that may be evaluated with the metrics available in that catalog.

2.1.1. Software metrics used in detection strategies

The software metrics applied in the definition of detection strategies are:

- DIT (Depth in Inheritance Tree): number of classes that are above a certain class in the inheritance hierarchy.

Table 1
Threshold catalog for object-oriented software metrics [15].

Metrics	Good/Frequent	Regular/Casual	Bad/Rare
CA	CA <= 7	7 < CA <= 39	CA > 39
CE	CE <= 6	6 < CE <= 16	CE > 16
MLOC	MLOC <= 10	10 < MLOC <= 30	MLOC > 30
NOC	NOC <= 11	11 < NOC <= 28	NOC > 28
NOF	NOF <= 3	3 < NOF <= 8	NOF > 8
NOM	NOM <= 8	8 < NOM <= 14	NOM > 14
NORM	NORM <= 2	2 < NORM <= 4	NORM > 4
NSC	NSC <= 1	1 < NSC <= 3	NSC > 3
NSF	NSF <= 1	1 < NSF <= 5	NSF > 5
NSM	NSM <= 1	1 < NSM <= 3	NSM > 3
PAR	PAR <= 2	2 < PAR <= 4	PAR > 4
SIX	SIX <= 0019	0019 < SIX <= 1333	SIX > 1333
VG	VG <= 2	2 < VG <= 4	VG > 4
WMC	WMC <= 11	11 < WMC <= 34	WMC > 34
DIT	DIT <= 2	2 < DIT <= 4	DIT > 4
LCOM	LCOM <= 0167	0167 < LCOM <= 0,725	LCOM > 0725
NBD	NBD <= 1	1 < NBD <= 3	NBD > 3
RMD	RMD <= 0467	0467 < RMD <= 0750	RMD > 0750

[1] http://pmd.sourceforge.net/.
[2] https://marketplace.eclipse.org/content/codepro-analytix.
[3] http://metrics.sourceforge.net/.
[4] http://www.sonarqube.org/.

- LCOM (Lack of Cohesion between Methods): measure of the internal cohesion of classes.
- MLOC (Method Lines of Code): number of lines of code in the method.
- NBD (Nested Blocks Depth): maximum number of nested blocks of a method.
- NSC (Number of Children): number of daughter classes regarding to a given class.
- NOF (Number of Fields): number of attributes of a class.
- NOM (Number of Methods): number of methods in a class.
- VG (McCabe's Complexity): cyclomatic complexity of a method.
- WMC (Weighted Methods per Class): total weight of a class regarding to the weights of each class method.
- SIX (Specialization Index): evaluates how much a subclass overrides the superclass.

2.1.2. *Threshold ranges used in bad smells detection strategies*

To filter the universe of entities present in software and thus enabling the identification of instances of bad smell, each software metric used by a detection strategy must be related to a threshold value. This threshold defines the range of values of interest relative to a specific software metric. Thus, it is possible to use such metrics as support to identify bad smell. For example, to find classes in a system that are very large in terms of code lines, LOC may be used as a support metric in a detection strategy filter. Additionally, it is necessary to establish a threshold for LOC so that the classes that override the range of values determined by the threshold may be identified correctly. An example of threshold is 1000, where LOC $>$ 1000 filter determines a minimum threshold for this metric to indicate large classes.

Our detection strategies of bad smells use three ranges of thresholds: Good/Frequent, Regular/Casual and Bad/Rare (see Table 1). Good/Frequent range addresses values with high frequency, characterizing the most common values for metric. Bad/Rare range addresses values with low frequencies, and Regular/Occasional range addresses values that are neither very frequent nor infrequent [15]. For Ferreira et al. [14], values considered frequent in systems indicate that they address the common practice in the development of high-quality software, which serves as a comparison parameter of software's. Uncommon values indicate unusual situations in practice, so, points to be considered critical, may be indicative of structural problems. In our strategies, we consider Good: upper limit of Good/Frequent range; Regular: lower limit of Regular/Occasional range; and Bad: lower limit of Bad/Rare range.

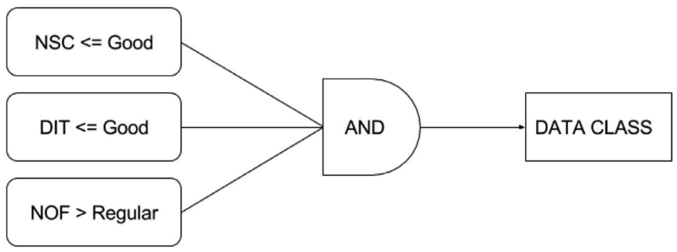
2.1.3. *Bad smells detection strategies*

To present the detection strategies for five of 22 bad smells proposed by Fowler [18] we use metrics and thresholds described by Filó et al. For each strategy, we show a brief description of the bad smell it refers to, the metrics, and the strategy itself.

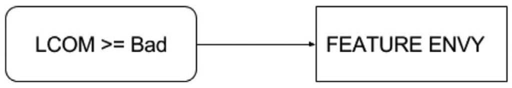
Data Class. Refers to classes containing attributes, getting and setting methods, and nothing else. These classes function as an aggregator usually without complex processing. The metrics composing this strategy detection are:

- NSC: since a Data Class mimics a primitive type of data, it is inferred that this type of class tends not to have subclasses. It is expected that a Data Class only provide access to its attributes, with get and set. Thus, we consider that low NSC values, along with other characteristics, help to identify Data Class.
- DIT: according the considerations made for NSC, it is assumed that a Data Class, in general, does not inherit responsibilities from superclass. Thus, it is considered that a low DIT, associated with other characteristics, may help to identify Data Class.
- NOF: since the class stores only data, we assume a high NOF indicates Data Class.

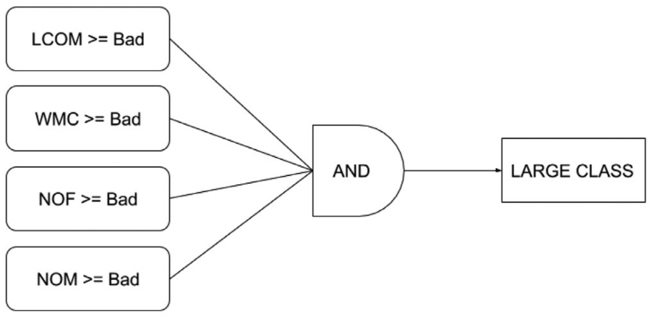
Fig. 1a illustrates the detection strategy for Data Class. NSC and DIT should have values within the range Good of the metric, including the



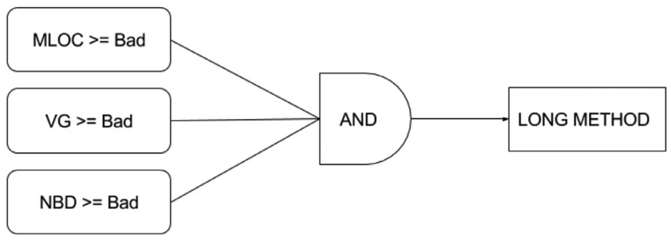
a: Data Class



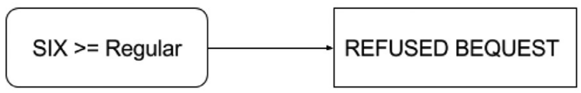
b: Feature Envy



c: Large Class



d: Long Method



e: Refused Bequest

Fig. 1. Detection strategies.

upper limit of that range; NOF value must be within the Regular or Bad intervals, including lower limit of the metric's Regular interval.

Feature envy. Refers to method that seems more interested in a class other than the one it actually is in. Fig. 1b shows its detection strategy. LCOM is the only metric in the catalog able to detect the interest of a class by data from another class.

Large class. Refers to classes that have many responsibilities. They concentrate the system intelligence by performing an excessive amount of work and becoming an aggregation of different abstractions. They are large, complex and have low cohesion. The detection strategy for Large Class, Fig. 1c, is composed by the metrics:

- NOF, NOM: a high amount of attributes (NOF) and methods (NOM) points to excessive knowledge and processing of the class.
- WMC: consider that the higher the weight of class, the greater the overhead of its operations. WMC may indicate many class responsibilities.

- LCOM: lack of cohesion between class methods indicates overload responsibilities. High LCOM suggests that class is a Large Class.

Long method. Refers to complex methods, too long and/or with various responsibilities, so it is difficult to understand, change or extend. The detection strategy for Long Method, Fig. 1d, is composed by the metrics:

- MLOC: long methods usually have an amount of lines of high code.
- VG: high cyclomatic complexity of method indicates high complexity of method.
- NBD: high block nesting value in method is an indication of complex/excessive processing.

Refused bequest. Refers to subclasses that have inherited methods and data of their parents, but they just use a few of them. Fig. 1e shows its detection strategy, which is composed by metric SIX. The greater the degree of overwriting that "daughter" class has in relation to the superclass, the greater the chances of the "daughter" class being refusing responsibilities provided by inheritance.

We did not define detection strategies for Fowler's bad smells: Duplicated Code, Message Chains, Parallel Inheritance Hierarchies, Speculative Generality, Switch Statements, Temporary Field, Primitive Obsession, Inappropriate Intimacy and Comments, because the metrics in the catalog used [15] do not support their detection.

Although the metrics used allow detection strategy's definition for the bad smells Alternative Classes with Different Interfaces and Incomplete Library Class, they were not considered, because we did not find tools to evaluate them.

The following bad smells were not included due to the reasons [13].

- Lazy Class: we found four tools for Lazy Class, but they are not available for download. True Refactor was available for download, but it could not be found.
- Middle Man: Code Bad Smell Detector is the only tool available to detect it. We could not run it due to problems in the configuration file.
- Data Clumps: there are five tools available for it, but InCode,[5] InFusion[6] and IntelliJ IDEA [16] need license to use. Code Bad Smell Detector presented a problem in the configuration file, and Stench Blossom [33] presented problems of incompatibility with the Eclipse's version used in this study.
- Long Parameter List: there are nine detection tools for this bad smell. However, five are unavailable for download. IntelliJ IDEA needs license; Gendarme does not perform detection on Java and, PMD presented faults in trying to export the results of some systems; Checkstyle[7] did not identify Long Parameter List for the selected systems.
- Shotgun Surgery: there are three tools unavailable for download. Among those, IntelliJ IDEA needs a license, and iPlasma[8] presented problems of exporting the results.

2.1.4. *Thresholds and automated detection of bad smells*

The identification of classes and methods with structural problems reflects the impaired software quality. The aim of this study is to evaluate the ability of thresholds metrics proposed by Filó et al. assisting the identification of Fowler's bad smells, and thus ensuring the quality of software. For that, RQ1 and RQ1.1 are investigated.

Results of the strategies proposed in Section 2.1.3 were compared with the results of JSpIRIT and JDeodorant tools. The results produced by these tools are not considered correct in this study, for there are no previous studies indicating the precision of such tools, even though they are used in studies regarding bad smells [13,15,40]. Therefore, the analysis made provides an overview of how the results of the strategies proposed are compared to those tools. We have used three evaluation metrics widely known for the analysis: Recall, Precision and *F*-measure [17,39,40,42]. These metrics were chosen for they are measures used to verify the effectiveness of results, besides being complementary.

Table 2
Systems selected from Qualitas.class Corpus 2013.

#	Systems	Size	No. of classes	No. of methods
1	aoi-2.8.1	9.4 MB	841	6915
2	checkstyle-5.6	53 MB	560	2896
3	cobertura-1.9.4.1	11 MB	174	3520
4	displaytag-1.2	10 MB	336	1728
5	itext-5.0.3	7.8 MB	587	5982
6	jedit-4.3.2	15 MB	1183	7315
7	joggplayer-1.1.4s	7.2M	363	2299
8	jsXe-04_beta	17 MB	270	1445
9	pmd-4.2.5	12 MB	914	5958
10	quartz-1.8.3	12 MB	316	2923
11	squirrel_sql-3.1.2	13 MB	169	689
12	webmail-0.7.10	12 MB	129	1091

JSpIRIT and JDeodorant are open-source code plugin of Eclipse that identifies bad smells in Java language [13,15,40]. JSpIRIT uses a metric-based approach, and was used in this experiment to identify: Large Class, Long Method, Data Class, Feature Envy and Refused Bequest. JDeodorant identifies: God Class (similar to Large Class), God Method (similar to Long Method), Feature Envy and Switch Statement. Detection techniques applied by JDeodorant are based on refactoring for God Class and Feature Envy and slicing techniques to detect God Method [17]. In this experiment, JDeodorant was used to detect Large Class, Long Method, and Feature Envy.

To implement the scripts with the detection strategies we have used FindSmells [47]. FindSmells is a bad smells detection tool that allows the user to select one or more files in XML format containing software measurements. It processes these files and inserts the measurements in a database. The user selects the system and the bad smell he wants to identify. After, FindSmells runs the strategy that was implemented for that bad smell. It filters out methods, classes, and packages that have anomalous measurements of OOS metrics in the context of software measurement process. Anomalous measurements are those that are significantly away from what is common, and may indicate problems with quality in the artifact measured. The detection tools and the proposed strategies were executed in 12 systems of small to medium size (see Table 2).

2.1.5. *Analysis with JSpIRIT tool*

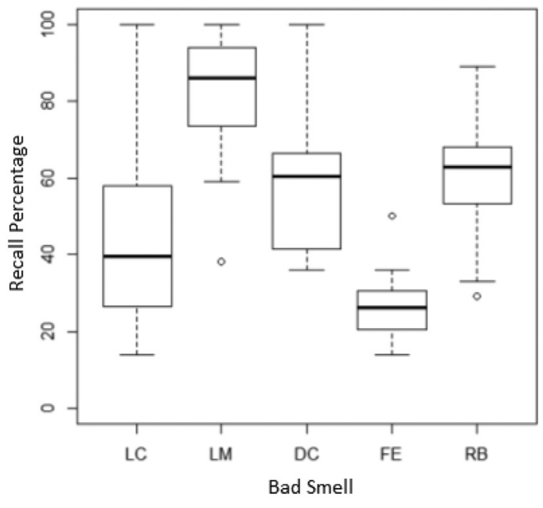
Results of the strategies proposed were compared with the results of JSPiRIT in terms of Recall, Precision and *F*-measure metrics. For Data Class, results were computed on the basis of eight of 12 systems since, for the other systems, JSpIRIT did not find instances of Data Class, or there was a memory limit exceeded by the tool.

2.1.5.1. Recall analysis. Recall measures the ability of detection strategies to find bad smells considering the software under review. The greater the value of Recall for a strategy, the greater is its capacity of finding instances of bad smell.

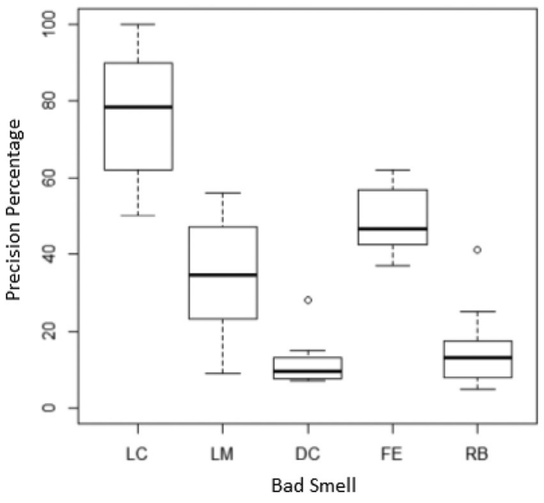
Fig. 2a shows the results of Recall distribution by bad smells. The medians of the Recall are greater than or equal to 40% for Large Class (LC), Long Method (LM), Data Class (DC), and Refused Bequest(RB). That happened because for six of 12 systems analyzed, the Recall was greater than or equal to 40% to each bad smell. This result is reasonable because it indicates that in an extensive universe of possible instances, the detection strategies were able to find a significant amount of them.

Recall distribution for Feature Envy has a median of 26%, and the third quartile, 75%, is not far away, with a value close to 30%. A justification is the lack of metrics adequate to detect Feature Envy in the

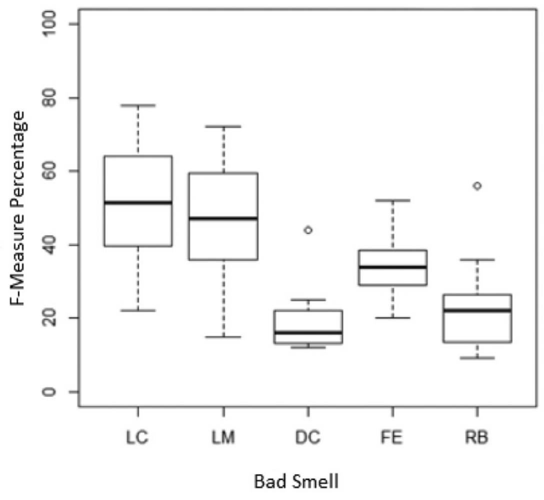
[5] http://www.intooitus.com/inCode.html.
[6] http://www.intooitus.com/inFusion.html.
[7] http://checkstyle.sourceforge.net/index.html.
[8] http://pmd.sourceforge.net/.



a: Recall distribution by bad smells



b: Precision distribution by bad smells



c: F-mesure distribution by bad smells

Fig. 2. Results of JSpIRIT.

catalog used [15]. We use LCOM. Another important fact, 100% Recall was obtained for at least one system, considering: Large Class, Long Method and Data Class. Refused Bequest, obtained Recall of 90% for at least one of the analyzed systems.

2.1.5.2. Precision analysis. Precision measures the ability of the strategies proposed to detect instances of bad smells. While Recall measures the number of results, Precision measures the quality of results. True instances in this study refer to those identified by JSpIRIT. However, we do not state that the data reported by this tool is indeed true. These results are used only for comparison with our strategies.

Fig. 2b presents the results of the precision for each of five bad smells analyzed. It shows that the medians are over 40% for Large Class and Feature Envy. When checking the data of the systems for such bad smells it is noted that six of 12 have Precision greater or equal to 40%. For Long Method, Precision is little over 30%.

2.1.5.3. F-measure analysis. *F*-measure metric verifies the precision of the results generated. It considers the results of Recall and Precision formulas, and may be interpreted as a weighted average between these two measures.

Fig. 2c shows the results of the *F*-measure distribution by each bad smell analyzed. The median obtained for *F*-measure is close to 50% only for Large Class and Long Method. That means, by balancing Recall and Precision measurements, there is a significant precision and, therefore, the detection strategies proposed for these bad smells proved to be compatible with the results of JSpIRIT. The median value of Feature Envy was not lower than the values for Large Class and Long Method. Data Class and Refused Bequest had even smaller results based on the set of selected systems.

Fig. 2c shows coefficients of up to 78% and 72% for Large Class and Long Method, respectively. *F*-measure was low for the others. In these cases, Recall was generally low, but Precision was significant, indicating that results generated by the strategies for these bad smells tend to differ from results of JSpIRIT. Noted that *F*-measure analysis may have been impacted by the fact that only eight of 12 systems had Data Class instances found by JSpIRIT.

2.1.5.4. Analysis for all bad smells. We also made a Recall, Precision and *F*-measure analysis for all bad smells, with the aim to compare the strategies proposed results with results of JSpIRIT. We found a median of 50% for Recall. It is reasonable to consider that even small systems may contain a high amount of bad smells, as showed in literature [29]. Recall was not lower than 14% for any of the bad smells rated. For 25% of the systems, Recall was significantly higher, with value $\geq$ 73%. Feature Envy was the only bad smell with Recall less than 40%. Regarding Precision, it got 41% for half of the systems. This result may be consequence of the low precisions obtained for Long Method, Data Class and Refused Bequest, with medians close to 30%, 10% and 15% respectively. The Precision values were inversely proportional to the values of Recall. *F*-measure distribution has smaller values compared to the distributions of Recall and Precision, respectively. This difference is justified by: for three bad smells, very low precision was obtained; we obtained low Recall for two bad smells. The *F*-measure ponders Recall and Precision in a single accuracy measure. Therefore, is expected a low concentration of *F*-measure values. All these analyses were done independently of the bad smell that the strategies are proposed to identify. This indicates that, in general, the results of the detection strategies proposed, when analyzed as a whole, have a low coincidence with the results of JSpIRIT. However, it is worthwhile to notice that this result does not reflect the situations of Recall and Precision analyzed in isolation.

2.1.5.5. Manual inspection of JSpIRIT results. Previous work has reported inefficiencies in tools for automated detection of bad smells in source code [6,13,28]. This may be caused, among other factors, by how the threshold and metrics used by tools are defined in the composition of

Table 3
Comparison of specialists results with detection strategies.

		Large class			Long method			Feature envy			Refused bequest		
		NV	V	DIF	NV	V	DIF	NV	V	DIF	NV	V	DIF
Squirel	Recall	100%	100%	0%	100%	100%	0%	50%	50%	0%	–	–	–
	Precision	60%	40%	−20%	17%	17%	0%	55%	15%	−40%	–	–	–
Webmail	Recall	15%	33%	18%	100%	100%	0%	27%	50%	23%	57%	80%	23%
	Precision	100%	50%	−50%	56%	41%	−15%	47%	24%	−23%	14%	14%	0%

detection strategies. These definitions may vary according to the context of the systems analyzed, and this may impact the quality of the results.

To minimize one of the main threat of this experiment – *the use of the results of JSpIRIT as a reference list of bad smells* – a validation of these results was conducted with the support of three specialists with knowledge in OOP and in the definitions of Fowler's bad smells. Each specialist analyzed the list returned by JSpIRIT and indicated for each bad smell, the true positives according to his knowledge. After, we compared the specialists' analysis and generated an oracle for this experiment. To an entity configure a bad smell in this oracle, it should have be indicated by at least two specialists as an anomaly. To allow manual analysis of instances of bad smells – based on the system source code – provided by JSpIRIT, the specialists chose from the analyzed set, Squirrel SQL and Webmail, to carried out the validation of their result.

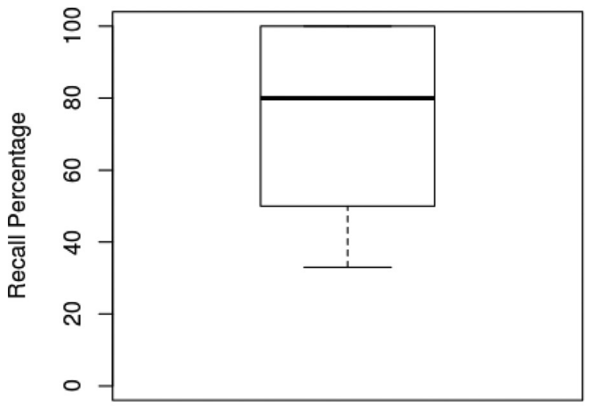
For comparison purposes, Table 3 shows Recall and Precision values for Non-Validated data (NV) and data after Validation by the specialists (V). It also displays the Difference (DIF) between Recall percentages and Precision validated and non-validated. Data Class was discarded from this analysis, because JSpIRIT did not return their instances for the systems evaluated. Although, JSpIRIT did not return instances of Refused Bequest for Squirrel SQL system, this bad smell was not discarded since JSpIRIT reported results for Webmail system.

DIF showed positive values for Recall regarding Large Class, Feature Envy and Refused Bequest, meaning that the validation of the specialists has identified false positives in the results generated by JSpIRIT. The removal of these instances identified by the specialists led to an increase in Recall. A negative DIF was also obtained for Precision in the case of Large Class, Long Method and Feature Envy. In this case, we conclude that, in some situations, the specialists did not identify instances in results generated by JSpIRIT. This fact had a negative impact on the Precision result. Analyzing Table 3 and the bad smells Large Class and Feature Envy, there is also an increase in the percentages of Recall accompanied by a decrease in Precision.

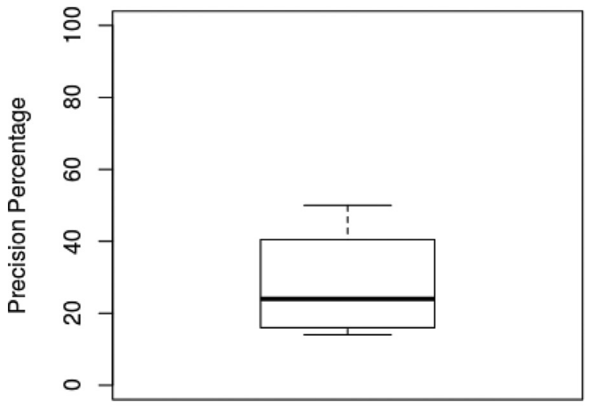
Fig. 3 shows Recall and Precision distributions for all bad smells. The specialists validated all instances of Squirrel SQL and Webmail systems. A median of 80% was obtained for Recall. They noted that 3/4 of Recall are greater than or equal to 55%.

We conclude that the participation of the specialists were positive, they have helped to improve the quality of referral lists provided by JSpIRIT. Results regarding Recall are quite expressive considering the difficulties inherent in the detection of bad smells in software. Even in small systems, there may be a high amount of instances of bad smells and, consequently, a high rate of recovery of instances is desirable. Results of Precision suggest either that the specialists have sometimes been unable to manually identify some valid instances of bad smells or that the strategies proposed indicate more instances of bad smells than actually exist.

Contrary to the observations made regarding Recall, the Precision distribution indicates a median of 24%, a low but expected value if we observe the high Recall concentration. The highest Precision value is 50%. This indicates that, in the best case, given the results of a detection strategy, half of the results provided are valid.



a: Recall distribution for all bad smells



b: Precision distribution for all bad smells

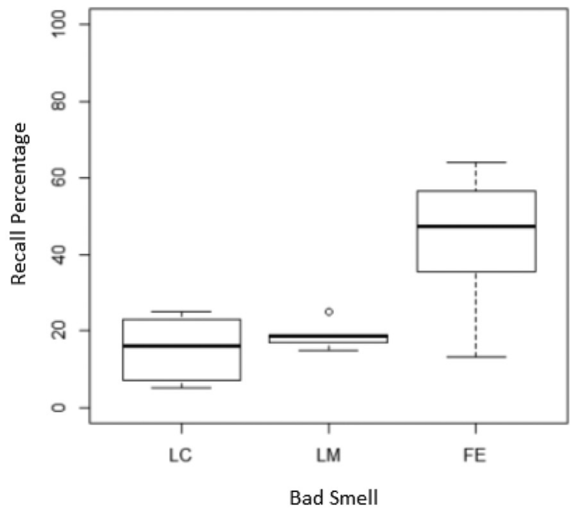
Fig. 3. Results of evaluated instances by the specialists.

2.1.6. Analysis with JDeodorant tool

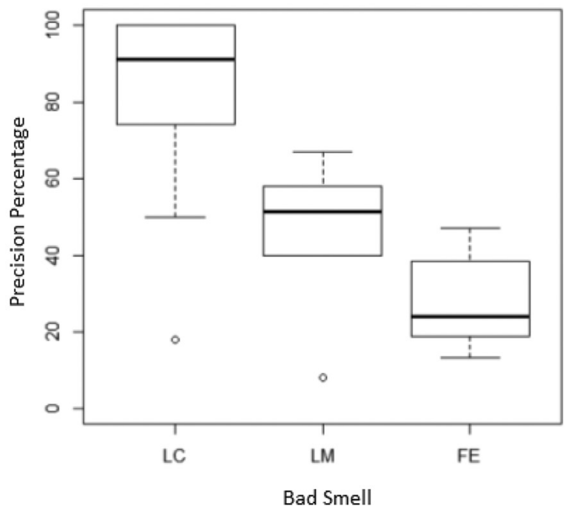
Results of the strategies showed in Section 2.1.3 were also compared with results of JDeodorant.

2.1.6.1. Recall Analysis. Figure 4a shows Recall distribution for each bad smells, taking into account the detection results provided by JDeodorant and the detection strategies presented in this study. This tool does not identify Data Class and Refused Bequest. So, both bad smells were discarded of this analysis.

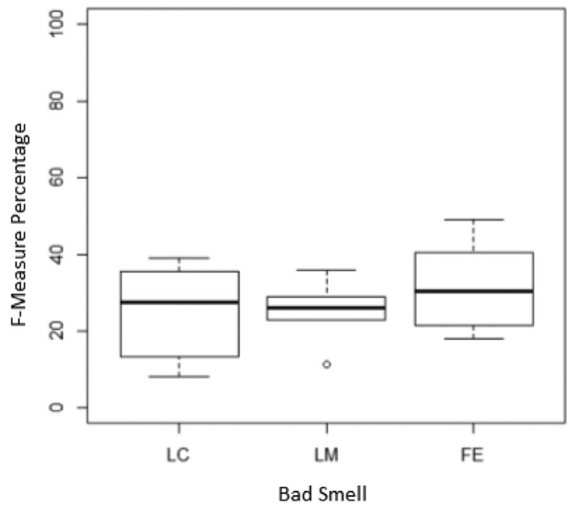
Regarding Large Class and Long Method, we observe a low concentration of Recall values for the detection strategies proposed according JDeodorant. However, when checking the reference lists, we observe that the amount of detection results reported by JDeodorant is, in general, larger than the quantity identified by the strategies proposed. For example, based on Checkstyle system, JDeodorant identified: 81 Large Class instances, against four instances reported by the respective detection strategy proposed, 95% more instances; and 110 instances of Long



a: Recall distribution by bad smell



b: Precision distribution by bad smell



c: F-mesure distribution by bad smell

Fig. 4. Results of JDeodorant.

Method, against 40 reported by the detection strategy of Long Method, 60% more instances. Additionally, for JsXe system: 29 Large Class instances, against four reported by the respective detection strategy, 86% more instances; and 126 instances of Long Method, against 52 reported by the detection strategy of Long Method, that is, 59% more instances.

The concentration of Recall for Feature Envy is in general moderate, with median 50%. Recall's highest value was around 65%. The first quartile is close to 40%, which means that, for 3/4 of the evaluated systems, Recall was significant, $\leq$ 40%. Contrary to what we observe in JSpIRIT analysis, Recall for Feature Envy was, to some extent, significant. To Fernandes et al. [13], the detection of bad smells implemented by JDeodorant is based on software metrics and Abstract Syntax Tree (AST). Such a hybrid approach may provide detection results more aligned to the results reported by the strategy proposed when compared to JSpIRIT.

2.1.6.2. Precision analysis. Precision measures the ability of the strategies proposed to detect instances of bad smells. While Recall measures the number of results, Precision measures the quality of results. True instances in this study refer to those identified by JSpIRIT. However, we do not state that the data reported by this tool is indeed true. These results are used only for comparison with the proposed strategies.

Fig. 4b shows the distribution of Precision for each bad smell, considering the detection results provided by JDeodorant and the detection strategies proposed. For Large Class and Long Method, we obtained a significant distribution of Precision values for the proposed strategies, in relation to JDeodorant. The medians are around 50% and 90%, respectively. For Large Class, 50% of the evaluated systems have more than 90% of Precision, reaching 100%. For Long Method, there is a consistency between the values obtained, which are between 40% and 70%. For Long Method, Precision presented a little discrepancy of values between the evaluated systems. We note that although Recall is small for both bad smells, the Precision is great. That is, although JDeodorant has returned a low amount of instances regarding the strategies, the result of the strategies is, in general, accurate.

For Feature Envy, we observe a low distribution of Precision values, with a median around 20%. This goes against Recall's significant results. A possible justification is that the greater the number of results retrieved by a detection strategy, the greater the chances of occurrence of false positives, as observed in the experiment with JSpIRIT.

2.1.6.3. Analysis of F-measure. Fig. 4c shows *F*-measure distribution results for each bad smells. None of the bad smells analyzed by JDeodorant obtained *F*-measure greater than 50% and the medians did not exceed 30%. Meaning that by balancing Recall and Precision, a low accuracy is obtained. Our detection strategies identified results close to those of JDeodorant.

Two factors may justify the disagreement between the results provided by JDeodorant and our detection strategies: the hybrid approach used by JDeodorant to detect bad smell; the difference between the results returned by JDeodorant regarding the strategies proposed. Considering JEdit system, JDeodorant identified 172 instances of Large Class, while the strategy proposed returned 34, a difference of approximately 80%. For Squirrel SQL, JDeodorant provided 21 instances of Large Class while only five were identified by the detection strategy. JDeodorant returned 40 instances of Long Method, whereas only 12 were provided by the strategy.

2.1.6.4. Analysis for all bad smells. Regarding the analysis for all bad smells we observe that the concentration of Recall values is, generally, low, although the results of Feature Envy have been significant. This is due to the fact that, for Large Class and Long Method, the Recall values are low enough to tend to the general distribution to the median of approximately 20%. These results indicate that the detection of bad smells displayed by the strategies proposed, concerning JDeodorant, is not very expressive. The distribution of Precision for all three bad smells identified by JDeodorant regarding the detection results of the strategies showed has, in general, high values, with median close to 50%. Results obtained individually for Feature Envy are low, but not enough to skew the overall distribution, since Large Class and Long Method exhibit a significantly high concentration of Precision. So, it has been shown that the accuracy of the strategies, in general, is expressive concerning JDeodorant. Given *F*-measure distribution, we observe a low concentration, with a median of approximately 30%, probably due to low Recall rates for Large Class and Long Method, as well as low Precision values from Feature Envy, where the highest value is 49%. That indicates, in general, the results of the detection strategies proposed, when analyzed as a whole, have a low coincidence with the results of JDeodorant.

2.2. Threats to validity

Construct threats. For this experiment, the software was randomly selected from Qualitas.class Corpus. This choice was based on the availability of downloadable source code and on the ability of these systems be analyzed by JSpIRIT and JDeodorant, which are not scalable for systems above 60 MB, 1K NOC and 7K NOM. This may have had a negative impact on the calculation of Recall, Precision and F-measure, for the larger the sample size, the greater its representativeness and diversity tend to be [34].

The detection strategies were proposed according to metrics and their thresholds available in Filó et al. catalog, and in Qualitas.class Corpus. The metrics and their thresholds may have had a negative impact on the results, since strategies may not have considered some aspects that characterize each bad smells. To mitigate this problem, each metric was carefully selected to compose the strategies and thresholds were defined according to the characteristics of each bad smell.

The quality of the detection results of JSpIRIT and JDeodorant is not guaranteed as said in Section 2.1.4. Using them as irrefutable reference of instances of bad smells in software may lead to an analysis with low reliability. To mitigate this problem, the results provided by these tools were rather only considered as a list for comparison purposes with the results generated by detection strategies. In our comparative studies, for three bad smells analyzed, the strategies proposed when compared with JSpIRIT presented medians of 80% and 24%, for Recall and Precision, respectively. When compared to JDeodorant, presented average Recall and Precision of 44% and 57%.

Fernandes et al. [13], in their study of bad smells tools, identify 84 tools. Results indicate that inFusion and PMD have the highest Recall, 50% and 67%, respectively. JSpIRIT and PMD have higher Precision, 80% and 100%, respectively. JDeodorant have 17% Precision and 33% Recall. But, they consider only Large Class and Long Method, and the analysis is performed with only small software. JDeodorant and inFusion presented the best results in the analysis regarding their ease of use.

Internal threats. Part of data collection analyzed has threats related to human factors. JSpIRIT and JDeodorant results are transferred manually for spreadsheets to make Recall, Precision and *F*-measure calculation feasible. The strategies proposed are implemented by FindSmells, which may contain errors. To mitigate these problems, FindSmells source code and the worksheets for calculating Recall, Precision, and F-measure have been validated by three OOP specialists. Beside that, a manual verification of the results has been performed to identify the intersection between the results of JSpIRIT and JDeodorant with the results identified by the detection strategies proposed.

To select the 12 systems to be evaluated, we tried to import 58 systems, but 46 were discarded due to: problems in importing the source-code into Eclipse IDE; or Eclipse IDE memory limit exceeded; or existence of more than one XML file with software metrics for the same system. The source-code importation of each system into Eclipse IDE development tool was necessary, because JSpIRIT and JDeodorant require systems to be imported for bad smells identification. Eventually, the lack of a proper import configuration may have led to the disposal of systems that could have been used in the study. To address this threat, several systems were submitted for importation in order to obtain a sufficient number of systems for analysis.

External threats. Qualitas.class Corpus has systems with more than one XML file of software metrics. Since it is not clear from the Corpus documentation if the files correspond to different packages or distinct versions from the same system, these systems were removed from the analysis as a precaution. Such a decision was taken to avoid possible conflicts between the content of the files, such as different values for the same metric.

Based on previous work investigating bad smells in source code [6,13,15], Recall, Precision and *F*-measure metrics are chosen to analyze the effectiveness of thresholds used through detection strategies present in this paper. Although there are other measurements interesting, the measurements chosen are effective to our research. Additionally, to validate the analysis performed, we had the support of three specialists.

Conclusion threats. Although, we consider that the 12 chosen systems are sufficient for the analysis proposed, this set may not be the reality of development in the context of large systems. Besides, all systems available in Qualitas.class Corpus are open-source and implemented in Java. This may also makes it difficult to generalize the results obtained to other development contexts, such as for other programming languages.

Due to technology limitations, such as memory limit when attempting to run large systems, the analysis of more software was infeasible. To address this threat, we select systems investigated in the literature with significant amounts of instances of bad smells, such as AOI, JEdit and Webmail [7].

3. Threshold value and bad smells manual detection

We analyzed Apache Maven, Version 3.0.5, with 18 MB, 864 classes and 6065 methods, to answer the RQ1.2. Maven is a tool for managing and understanding software systems projects [50].

3.1. Identification of bad smells by the specialists

Like the other experiments, the system source code was imported into Eclipse IDE, from which three specialists generated a list of bad smell's instances. They considered Large Class, Long Method, Data Class and Refused Bequest, whose brief descriptions appear in Section 2.1.3. To identify each type of bad smell, the specialists considered the detection criteria next. Comments and interfaces were ignored whenever present. For Large Class, class methods need to be more complex than simple gets and sets. For Long Method, the method must be also complex. Class containing, besides – constructors, gets and sets – other types of methods, these one should be considered as “simple” processing for the class to be designated as Data Class. For Refused Bequest, the specialists also considered that class contains one considerable number of new methods. Overlapped abstract superclass methods had no weight when considering Refused Bequest.

3.2. Results of bad smells manual detection

Results for each bad smell manually evaluated by the specialists in Maven system were compared with data obtained by the detection strategies proposed. Recall (R), Precision (P), and *F*-measure (F-M) metrics were used in this evaluation, as showed follows. It was also calculated the mean and standard deviation of these values for the discussion next.

	R	P	F-M
Large class	70%	37%	48%
Long method	73%	19%	30%
Data class	47%	32%	38%
Refused bequest	100%	2%	4%
Mean	72,5%	22,5%	30%
Deviation	21,7%	15,6%	18,8%

Analyzing these results, we have obtained a Recall mean of approximately 73%, taking into account all four bad smells evaluated. This value indicates that detection strategies, in general, were able to report a significant number of instances of bad smells regarding to those identified by the specialists. The average Precision obtained was approximately 23%, much lower than the Recall percentage identified. This means that although the number of instances recovered is expressive, the amount of false positive is high, that is, the number of valid instances from the specialists' point of view is low. Consequently, this discrepancy negatively affected the *F*-measure coefficient, which balance Recall and Precision, presenting an average value of 30%.

3.3. *Threats to validity*

Construct threats. This experiment used only Maven system, what may have impacted the results of Recall, Precision and *F*-measure. Regarding detection strategies, we use those presented in Section 2.1.3. So, threats to validity of this experiment are the same as those one.

Internal threats. Data collection also has threats related to human factors, because, Maven analysis was performed manually by three specialists in the source code. Such threat may be related to the complexity of the task performed manually, as well as the limitations time to complete the study. To mitigate it, we gave two weeks for the analysis, considering the size of the system and the number of bad smells to be identified.

The specialists used arbitrary thresholds in the criteria proposed to detect bad smells. Additionally, they applied more rigorous concepts than those defined by Fowler in the definition of Large Class, Refused Bequest, Long Method and Data Class. These factors constitute threats to the validity of the results of this study.

External threat. As previous experiments reported, the set of measurements chosen to evaluate the results of this study is Recall, Precision and *F*-measure. So, the same threats to external validity of Section 2.2, referring to the calculation of these metrics, apply here as well.

Conclusion threat. Considering that the specialists evaluated only the Maven system, it is not possible to generalize the results.

4. Software metric thresholds and fault prediction

This section presents an empirical study and their results regarding the experiments carried out to investigate the RQ2.

For each class-level software metrics: DIT, LCOM, NOF, NOM, NORM, NSC, NSF, NSM, SIX, and WMC, we investigate whether the value-ranges Regular/Occasional and Bad/Rare specified for thresholds [15] are sufficient indicators of software system faults.

4.1. *Study design*

We have inspired on the work of Wohlin et al. [54] to define the steps and execution of this study to answers RQ2. The steps are:

Step 1. *Select a threshold catalog.* As in the previous study reported in this paper, this experiment uses the catalog of threshold of Filó et al., (Section 2.1).

Step 2. *Select a tool to collect fault data.* We use BugMaps to collect fault data stored in bug-tracking systems [10]. BugMaps provides historical fault data for systems in two repositories, Bugzilla and Jira. We choose these tools based in the result of Januário and Ferreira's work [21]. These authors have compared data collection tools of software faults, and have concluded that BugMaps is the only tool that collects fault data in each class of software, exporting the result on CSV format.

Step 3. *Select the software systems for analysis.* We also use as basis Qualitas.class Corpus [50] to select the software to be analyzed. They were selected considering the availability of faults registered in Bugzilla and Jira. Table 4 exhibits these systems,

Table 4
Qualitas.class Corpus 2013 selected systems for threshold x fault study.

#	Systems	Size	# of Analysed class
1	ant-1.8.2	37,1 MB	869
2	aspectj-1.6.9	61 MB	1507
3	batik-1.7	94 MB	1473
4	cayenne-3.0.1	7,1 MB	1844
5	derby-10.9.1.0	73 MB	1851
6	jmeter-2.5.1	22 MB	977
7	maven-3.0.5	18 MB	696
8	myfaces-2.1.10	83 MB	1189
9	poi-3.6	49 MB	2149
10	roller-5.0.1	133 MB	500

where "Systems" refers to their names and versions; and "# of analyzed class" refers to the number of system classes associated with faults registered in BugMaps. Step 4 describes the process for obtaining this data.

Step 4. *Extract fault data from the selected software systems.* To do that, (I) data is collected in Bugzilla and Jira. This resulted on a CSV file, format required to accomplish data entry in BugMaps. (II) BugMaps was then informed of the desired time to filter the faults. Considering that the metrics of Qualitas.class Corpus refer to 2013, the fixed period comprises 2013–2016. (III) From the ID of each fault in the repository of the project sources (GitHub), BugMaps performs an association of faults with system classes and exports the data to a file on CSV format.

Step 5. *Classify measurements according to threshold ranges.* We perform an analysis of the metrics file of Qualitas.class Corpus for all systems selected on Step 3 to identify in which range (Good, Regular or Bad) the value of each metric, at the class level, is related with the thresholds used, [15]. Through scripts implemented in Java, the value of the metric has been replaced by the name of the range to which it corresponds.

Step 6. *Analyze the relation between threshold and faults.* We consider the output file generated by BugMaps containing the relation of the classes per system and their respective number of faults; and, the file of metrics obtained from the Qualitas.class Corpus containing the range of the metrics according to the catalog used [15]. These two files are combined to facilitate the analysis of the relationship between thresholds of software metrics and occurrence of faults in each class. The analysis is achieved based on descriptive statistics to investigate the relationship between two qualitative variables: "the range that each metric value refers to" and "the presence of faults in each class". To do that, we use dynamic tables that cross the investigated variables and calculate the frequency of occurrence of each crossing. The results are then converted to percent and the mean and the standard deviation are calculated.

4.2. *Results of the analysis of software metrics by system*

Table 5 shows by metric, and by system, the percentages of classes with faults classified in the Good, Regular, and Bad ranges, according to the threshold used. The complement of the value of each cell, in relation to 100%, corresponds to percentage of classes that did not present faults classified in the respective ranges of values. Observing JMeter, 83.93% of its classes with NOF metric in Bad range shows fault. Results exhibited in this table allow us to analyze which values ranges are best indicators of the occurrence of fault per metric.

DIT. For 50% of systems, DIT obtains the highest percentage of classes with faults in Good range, with approximately 23% of classes in this range. For Regular range 40% of systems obtain the highest percentage of classes with faults, average of 19% of classes, against 10% of the system in Bad range. Merging the result of Regular and Bad ranges,

Table 5
Percentage of classes with faults per system for each software metric.

Metrics	Band	Ant	AspectJ	Batik	Caienne	Derby	Jmeter	Maven	MyFaces	POI	Roller	Average
		%	%	%	%	%	%	%	%	%	%	%
DIT	Good	20,10	9,50	8,46	18,45	25,35	61,56	20,37	0,42	45,41	15,61	22,52
	Regular	17,35	6,78	14,52	21,54	33,98	42,76	2,25	1,12	27,30	21,70	18,93
	Bad	8,86	3,85	24,51	3,60	29,58	54,73	0,00	0,00	21,21	0,00	14,63
LCOM	Good	10,75	4,95	9,56	14,48	18,42	48,52	13,32	0,28	34,45	15,00	16,97
	Regular	11,54	10,84	9,16	17,84	29,93	70,73	19,35	0,00	44,22	14,89	22,85
	Bad	33,20	17,14	20,48	30,61	53,02	74,44	34,52	3,41	48,78	20,69	33,63
NOF	Good	10,17	5,54	10,35	14,77	20,05	49,43	13,82	0,24	36,72	14,33	17,54
	Regular	22,22	15,38	11,59	25,00	36,79	74,75	28,57	1,22	49,71	21,51	28,67
	Bad	50,51	22,08	15,19	36,00	54,59	83,93	50,00	6,67	40,00	22,86	38,17
NOM	Good	9,68	3,25	7,45	12,50	18,93	47,22	14,00	0,27	33,65	14,04	16,10
	Regular	20,14	8,55	22,04	24,84	24,59	64,15	20,27	0,00	41,50	15,48	24,16
	Bad	40,24	21,13	18,30	37,78	52,59	79,23	36,36	2,83	55,14	24,47	36,81
NORM	Good	17,36	7,91	9,59	15,65	25,36	55,96	17,34	0,44	38,46	14,29	20,25
	Regular	32,00	11,49	27,50	27,38	34,67	63,64	25,00	0,00	51,61	28,81	30,21
	Bad	22,22	11,76	36,11	60,00	43,64	33,33	0,00	2,70	80,00	40,00	32,98
NSC	Good	18,61	7,81	10,43	15,38	27,85	57,64	17,24	0,63	38,63	16,82	21,10
	Regular	16,25	9,46	13,77	24,27	25,16	48,57	18,33	0,00	45,65	11,11	21,26
	Bad	11,54	16,36	10,98	28,72	23,68	36,59	12,50	0,00	34,09	8,33	18,28
NSF	Good	11,97	6,09	10,72	16,33	24,50	41,26	16,37	0,41	38,26	12,89	17,88
	Regular	29,13	15,31	10,29	25,77	37,70	67,14	36,36	0,56	37,16	28,21	28,76
	Bad	41,57	21,43	14,67	21,62	38,60	83,96	33,33	1,85	47,62	50,00	35,47
NSM	Good	15,45	7,24	10,40	17,28	25,35	52,33	15,91	0,47	34,90	15,17	19,45
	Regular	32,14	14,29	24,07	7,89	40,79	92,86	25,00	1,82	57,14	18,18	31,42
	Bad	51,16	17,65	8,77	34,78	46,67	88,89	52,94	0,00	70,97	37,50	40,93
SIX	Good	16,38	6,65	7,40	14,83	25,38	54,15	18,20	0,39	41,85	12,87	19,81
	Regular	21,55	13,46	17,48	20,28	32,08	68,22	13,64	1,52	33,54	22,94	24,47
	Bad	9,52	1,92	34,25	25,00	27,44	30,77	0,00	0,00	25,00	33,33	18,72
WMC	Good	7.07	1,87	5,04	9,87	12,21	38,71	11,28	0,00	27,24	7,37	12,06
	Regular	15,73	7,35	16,58	21,27	26,61	70,27	23,36	0,45	45,00	17,24	24,39
	Bad	44,25	23,40	22,60	47,62	56,35	91,21	46,30	3,17	71,23	39,24	44,54

which are the most critical, we obtain a result equivalent to that given in the Good range. According to Filó et al., Good range represents the most frequent values that possibly indicate good programming practices. Result suggests that DIT threshold may not be considered as good indicators of software faults.

LCOM. The percentage of classes with faults in Good or Regular ranges, on average, is 14% and 18%, respectively. For Bad range 100% of the systems have the highest percentage of classes with faults. The average percentage corresponds approximately 29% of classes. This value is up to 15% greater than that obtained in previous ranges. So, the Bad range of LCOM metric proves to be effective in the indication of occurrence of faults in systems.

NOF. The average percentage of classes with faults in Good range is approximately 18%. Considering Regular and Bad ranges, respectively, 10% and 90% of 10 systems present the highest percentage of classes with faults. The mean percentage corresponds approximately 29% for Regular range and 38% for Bad range. Therefore, Bad range for NOF proves to be effective in indicating system faults.

NOM. The average percentage of classes with faults in Good range is approximately 16%. Considering Regular and Bad ranges, respectively, 10% and 90% of 10 systems present highest percentage of classes with faults. The mean percentage corresponds approximately 24% for Regular range and 37% for Bad range. So, we conclude that Bad range for NOM is effective in indicating faults.

NORM. The average percentage of classes with faults in Good range is approximately 20%. For Regular and Bad ranges, respectively, 30% and 70% of 10 systems present the highest percentage of classes with faults. On average, these ranges obtain approximately 30% for Regular range and 33% for Bad range. We conclude that Bad range for NORM is a good indicator of system faults.

NSC. For 50% of the systems, this metric obtains the highest percentage of classes with faults in Good range, corresponding, on average, approximately 21% of the classes in this range. For Regular range, 20% of the systems obtain the highest percentage of classes with faults, average of 19% of classes, while 30% of the systems have the highest percentages in Bad range. Adding up the result of the most critical ranges, Regular and Bad, the total number of systems equals the result of Good range. We conclude that thresholds used for NSC are not good indicators of system faults, although their catalog presents Good range of this metric as an indicator of good programming practices.

NSF. The average percentage of classes with faults in Good range is approximately 18%. Regarding Regular and Bad ranges, respectively, 20% and 80% of these systems present the highest percentage of classes with faults. The respective average percentages of classes with faults are approximately 29% for Regular range, and 35% for Bad range. Therefore, results suggest that NSF threshold are useful in predicting system faults.

NSM. The average percentage of classes with faults in Good range is approximately 19%. Regular and Bad ranges have, respectively, for these systems 20% and 80%, the highest percentage of classes with faults. On average, the percentage of classes with fault is approximately 31% and 41% for Regular and Bad ranges, respectively. The results suggest that NSM is useful in predicting software faults.

SIX. Two of 10 systems present the highest percentage of classes with faults in Good range. On average, we observe 20% of classes with faults in this range. Regarding Regular and Bad ranges, respectively, 20% and 80% of 10 systems present the highest percentage of classes with faults. Their respective average percentages of classes with faults are approximately 24% for Regular range and 19% for Bad range. There is an increase in the result from Good to Regular, but a decrease from Regular to Bad. Therefore, for SIX, Regular range is the most effective in indicating system faults.

WMC. None of the 10 evaluated systems present for WMC the highest percentage of classes with faults in Good or Regular range. The approximate average percentage of classes with fault in these ranges is 12% and 24%, respectively. For 100% of the systems, the highest percentage of classes with faults is classified in Bad range, with an average percentage of approximately 45% of classes. This value is 33% higher than that obtained in the previous ranges. Therefore, Bad range for WMC is able to indicate the largest number of system faults.

These results show that, for seven of 10 software metrics analyzed – LCOM, NOF, NOM, NORM, NSF, NSM and WMC – the Bad range is effective to indicate faults. None of 10 systems obtained the highest percentage of classes with faults in Good range for the metrics: NSM, NSF, NORM, NOM and NOF. For NOF, NOM and NSF, the percentage of class per range increases as the criticality of the range increases from Good to Bad, i.e., the more critical the range, the greater the percentage of class with faults. Regarding SIX, Regular range is the most effective in the indication of faults. For DIT and NSC metrics, it was not possible to state that their thresholds are indicators of system faults, both are metrics related to the application of inheritance. Result indicates that inheritance level is not good predictor for occurrence of software faults. Thus, the study confirms that in general the thresholds used [15] may support the prediction of faults in software systems.

4.3. *Results of the analysis by software metric independent of the system*

To complement the study of Section 4.2, we provide a general analysis per metric independent of the source system. In this analysis, the data set of all classes from the 10 systems evaluated is consolidated into a file, from which, for each metric, the frequency of classes with presence or absence of faults in the ranges Good, Regular, and Bad is calculated. This is done in order to verify the relation between the percentages of classes classified on the threshold values ranges, and the presence of faults. The RQ2 is answered considering the data set of all these systems. Table 6 presents, for each metric, the percentages of classes with rated faults in each of this value range. The complement of each cell, regarding to 100%, corresponds to the percentage of classes that do not present faults classified in the respective value range. The mean and the standard deviation are calculated for each of the three ranges, allowing us to analyze the ranges considered the best indicators of faults occurrence.

The results obtained for each metric here are compatible with those one observed when each individual software was analyzed, where seven of 10 metrics evaluated had the highest percentage in classes with faults on the Bad range.

Regarding value range Good, only two of 10 metrics analyzed, DIT and NSC, presented highest percentage of classes with faults. On average, this range obtained approximately 18% of classes with faults. This result is consistent with the definition present in the catalog used [15], where Good range corresponds to values that presented high frequency in the system. Although these values do not necessarily express the best practices in Software Engineering, they may indicate a standard quality and, therefore, entities so classified tend not to fail.

Considering Regular range, only SIX presents the highest percentage of classes with faults. However, on average, we obtained approximately 26% of classes with faults, considering all 10 metrics under analysis.

Table 6
Percentage of classes that presented faults by range and by metric.

Metrics	Good	Regular	Bad
	%	%	%
DIT	2100	2088	2073
LCOM	1639	2361	3508
NOF	1709	2959	4153
NOM	1522	2482	3977
NORM	2010	2879	3121
NSC	2117	2069	1817
NSF	1752	3034	3931
NSM	1926	3311	3918
SIX	1945	2510	2143
WMC	1157	2473	4437
Average	1788	2617	3308
Standard deviation	2,98	4,13	9,65

This is a relatively significant value. It is 8% higher than the value obtained for the Good range. This percentage's increase of classes identified with faults is expected, because the Regular range corresponds to values that are not very frequent, nor rare. Classes having measures considered Regular are more likely to present problems regarding the value of Good range (Table 6).

Regarding Bad range, 70% of the metrics analyzed, LCOM, NOF, NOM, NORM, NSF, NSM, and WMC obtain highest percentage of classes with faults. The average percentage corresponds to approximately 33% of classes. This value is up to 15% higher than that obtained on Good and Regular ranges. Result is in line with the threshold catalog used, which describes Bad range as low-frequency values, being the most critical in terms of the propensity for problems to occur. Therefore, the application of Bad range is effective to indicate fault occurrence in future versions of the evaluated systems.

Results obtained suggest that the thresholds proposed by [15] are useful to predict faults in software systems. Thus, the RQ2 response is affirmative.

4.4. *Threats to validity*

Construct threats. We have selected BugMaps tool to support the collection of faults in systems available at BugZilla and Jira repositories. BugMaps is an automated tool, so, it may contain glitches. Nevertheless, its choice was based on results of previous work that indicate its applicability and effectiveness [10,21].

Another threat is related to reliability of BugZilla and Jira, although both are used in software factories. Also we implemented a Java routine to automate the process for classifying the values of the metrics threshold according to Good, Regular, and Bad ranges. This routine may contain errors that may impact the results. To mitigate it, this routine was tested on a small dataset before being used in this study.

Internal threat. During data collection, some threats related to human factors may have affected the process. At the cross-referencing fault data with the classification of software metrics by ranges Good, Regular and Bad, the classes with faults that had no software metrics computed in Qualitas.class Corpus were removed manually. This was necessary because the corpus contains information regarding 2013 version of the systems, while the fault data covers information from 2013 to 2016 versions.

External threat. We used systems implemented in Java and cataloged by Qualitas.class and discarded the ones whose fault data were not available in Bugzilla or Jira repositories, totaling 10 analyzed systems. Although this number has been considered sufficient, the conclusions obtained may not be generalized to any development context. For example, systems implemented in other programming languages or with divergent characteristics regarding the systems analyzed.

Conclusion threats. The analysis done is based on the percentage of classes with faults that are classified in the Good, Regular, and Bad value ranges.To support the investigation of RQ2, the mean and the standard deviation were computed for metrics known in the literature. Although there are other ways of analyzing this type of study, the analysis by means of these metrics was considered sufficient. Additionally, the entire analysis was computed using spreadsheets. To minimize possible problems, a manual inspection of the generated files was performed. It is emphasized that in this study only the presence or absence of faults in the class is considered and not the number of faults identified by class. Although we consider that the number of faults justify conducting a complementary study, we consider that our results provide sufficient evidence for the established objectives.

5. Related work

This section reports how researchers are investigating the application of metrics thresholds to detect bad smells and to predict fault in OO software systems. It shows that several approaches are available for bad smells detection. Basically most of them proposed a set of rules, which are combinations of software metrics with their thresholds [3,14,15,25,26,30–32,38,42,43,45,46,48,53].

Sahin et al. [42] use the generation of rules for bad smell detection as a two-level optimization problem, where top-level generates a set of detection rules and the lower level generates artificial bad smells. The threshold for each metric is done manually. Results show seven types of code smells detected with an average of over 86% in terms of Precision and Recall.

For Singh and Kahlon [46] the propensity of a module to have fault is somehow associated with a bad design. They address this problem using risk analysis at five different levels. They used three versions of Mozilla Firefox as dataset. Results show that some metrics have threshold for various levels of risk that are of practical use in predicting classes with faults.

Vale et al. [53] compare three methods [3,26,38] to derive thresholds in a benchmark of Software Product Lines (SPL). They evaluated which of these methods provide appropriate thresholds to four metrics used in SPL: Lines of Code, Coupling between Objects Classes, Weighted Method per Class, and Number of Constant Refinements. The thresholds obtained identified God Class.

Marinescu [30] proposes one of the first works for detecting bad smells via software metrics. To find threshold, he uses two statistical means: average to determine the most typical threshold of the dataset, and standard deviation to obtain a measure of how much the thresholds in the data are scattered. With thresholds, he measure 45 Java and 37 C+ + systems regarding the metrics: Number of Methods, Lines of Code and Cyclomatic Number. The thresholds applied in our work were derived considering the characteristics of distributions [15].

Others, like Zhang et al. [57], argue that software metrics lack precision in detecting bad smells and propose a code pattern-based approach whose aim is to define bad smells as patterns of source code, so that bad smells may be detected through examining these patterns in source code.

Zhang et al. [60] introduce an overview about an empirical study to investigate the impact of bad smells in terms of their relationship with faults.

Zhang et al. [59] conduct a SLR with 319 papers from 2000 to June 2009 to identify the knowledge about bad smells. They analyze in detail 39 papers. Results indicate that the level of knowledge varies for different bad smells. Duplicated Code get the most attention of researchers, whereas, Message Chains, have very few studies. Study of Feature Envy, Long Method and Large Class focuses on improving the understanding. The focus of other bad smells is on developing tools and methods to automatically detect them. Only five studies directly investigate the impact of using bad smells, four of them indicate that not all bad smells have negative impact on software, Duplicated Code for example may increase the reliability of software. Data Class, Refused Bequest and Feature Envy are not significantly associated with the specific severity level of software faults. Most of the studies are based on open source project, expert opinion are infrequently used. Results suggest that the studies mainly focus on objective data, while subjective data are rarely used.

Nascimento and Sant'Anna [36] also investigate the relationship between bad smells and software faults. Their results show that classes with bad smells, in most cases, are more related to occurrence of faults. God Class and Feature Envy were the ones with the highest proportion of faults on their classes.

Macia et al. [29] study the impact of code anomalies in architectural design degradation. They use MuLATo [3], Together [4] and Understand [5] tools to collect metrics for the detection strategy. Results show that most architectural problems in source code emerge from an anomalous code.

There are several techniques to predict fault in software systems: machine learning [22,44]; historical analysis of the systems [2,19,24,44]; design patterns application [12]; source code statistical analysis [61]; metric-based defect prediction [1,22,23,55,56].

Zhang et al. [58] study the relationship between bad smells and fault in source code file level. They capture the bad smells from each collected source code files, and the number of faults associated with it. They consider: Duplicate Code, Data Clumps, Switch Statements, Speculative Generality, Message Chains, and Middle Man. They also investigate if their results may be used to emphasize refactoring. Results suggest that source codes containing Duplicated Code, and Message Chains are associated with much higher number of faults, therefore should be prioritized for refactoring. The remaining ones are not likely to be fault prone.

Thresholds of software metrics are also used in predicting faults. Gyimothy et al. [20] describe how fault prediction of Mozilla may be made, using metrics CK, [9]. Catal and Diri [8] propose a grouping of metric threshold to solve the problem of predicting faults when the fault labels for modules are unavailable.

Lavazza and Morasca [27] investigate the consequences of defining thresholds on internal measures without taking into account the external measures that quantify qualities of practical interest. They focus on fault-proneness as the specific quality of practical interest. Results indicate that distribution-based thresholds appear to be unreliable in providing sensible indications about the quality of software modules.

Morasca and Lavazza [31] propose and evaluate an approach based on the existence of a statistically significant model that relates a given measure to fault-proneness, defined as the probability that a module contains at least one fault.

Morasca and Lavazza [32] extend this former work, introducing four ways for setting thresholds on a given measure *X*. They use the value of *X* where a fault-proneness model curve changes direction the most. Then, they use the values of *X* where the slope has specific values. They apply their approach to data from the PROMISE repository by building Binary Logistic and Probit regression fault-proneness models. Results show that their thresholds effectively detect "early symptoms" of module faultiness, and achieve a level of accuracy in classifying faulty modules that are close to usual thresholds' fault-proneness.

Lavazza and Morasca [27], Morasca and Lavazza [31,32] propose, and evaluate metric thresholds based on fault-proneness, whereas Filó's and Ferreira's approaches are based in benchmark data. The main contribution of the present paper is to evaluate Filó's catalogue considering as the main points: it is a catalogue big, 18 metrics; previously evaluated; and previous results [14] indicate that thresholds gathered, via benchmark data, are applicable regardless the size, type, or application domain of systems.

This paper extends our previously work [48], which proposes detection strategies for five bad smells based on thresholds [15], and preliminarily shows the study done to investigate the efficacy of these thresholds in identifying bad smells in 12 systems of Qualitas.class Corpus [50]. In that work, we use reference lists provided by the bad smells detection tools, JSpIRIT [52] and JDeodorant [51] and concluded that the results of the detection strategies are not dissimilar to these tools, which identify bad smell automatically. We also observe that the results of the detection strategies are closer to the results of JSpIRIT than those of JDeodorant.

Several works propose thresholds, (Section 1), but they are not evaluated in general. Our research contributes with an investigation not yet considered: the usefulness of software metric thresholds to bad smell detection and fault prediction.

6. Conclusion

This paper reports the results of the research done to evaluate the usefulness of software metrics thresholds in the identification of bad smells in source code and in predicting software system faults. The motivation is due to the fact that, even though threshold definition for software metrics has been studied in the literature, there is still no in-depth evaluation of the proposed thresholds. We have used the metric threshold's catalog defined by Filó et al. [15]], which has the largest quantity of metrics with thresholds, 18, and has been partially evaluated previously, [48].

To evaluate the usefulness of these thresholds, we have conducted two studies. The first one investigates the effectiveness of thresholds to identify Large Class, Long Method, Data Class, Feature Envy and Refused Bequest bad smells. The second study investigates the use of thresholds in predicting faults according to Good, Regular and Bad range. Both studies are addressed to object oriented systems written in Java. The results of these studies show that thresholds are significantly effective in supporting the detection of bad smells, with generally high percentages for Recall and moderate for Precision and *F*-measure. This positive result was obtained from reference lists provided by bad smells detection tools, and from the lists generated manually by specialists. It also shows that thresholds are effective in predicting faults for most of the software metrics analyzed. The Bad range presents the highest percentage of classes with faults in the evaluated systems. We observe that the greater the criticality of the range of thresholds, the greater the percentage of classes with faults. The main conclusion of our research is that thresholds of metrics may be useful instruments in evaluating software quality. Identifying classes whose metrics are rated in Bad range may help developers to focus their efforts on classes that tend to fail, thereby minimizing the occurrence of future system problems.

As future works we consider the evaluations of the proposed strategies in other systems; manual evaluation of the 12 systems considered in this article; extension of the analysis done here to systems implemented in other programming languages; replication of these studies with other catalogs of thresholds to evaluate their applicability.

Declaration of Competing Interest

None.

Acknowledgments

Sponsors: CAPES, UFMG, CEFET-MG, CNPq.

We thank the specialists with knowledge in OO and Fowler's bad smells definitions: Eduardo Fernandes, Bruno Sousa, Mivian Ferreira, who help us doing the manual revision of three systems.

References

[1] G. Abaei, A. Selamat, H. Fujita, An empirical study based on semi-supervised hybrid self-organizing map for software fault prediction, Knowl. Based Syst. 74 (2015) 28–39.

[2] J. Al Dallal, S. Morasca, Investigating the impact of fault data completeness over time on predicting class fault-proneness, Inf. Softw. Technol. 95 (2017) 87–105, doi:10.1016/j.infsof.2017.11.001.

[3] T.L. Alves, C. Ypma, J. Visser, Deriving metric thresholds from benchmark data, in: International Conference on Software Maintenance, IEEE, 2010, p. 10.

[4] S. Artzi, S. Kim, M.D. Ernst, Recrash: making software failures reproducible by preserving object states, in: Proceedings of the 22nd European Object-Oriented Programming Conference, 2008, pp. 542–565.

[5] V. Basili, L. Briand, W. Melo, A validation of object-oriented design metrics as quality indicators, in: IEEE Transactions on Software Engineering, 1996, pp. 751–761.

[6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, E. Merlo, Comparison and evaluation of clone detection tools, Trans. Softw. Eng. 33 (9) (2007) 577–591.

[7] B. Cardoso, E. Figueiredo, Co-occurrence of design patterns and bad smells in software systems: an exploratory study, in: Proceedings of SBSI, 46, 2015, pp. 347–354.

[8] C. Catal, B. Diri, A systematic review of software fault prediction studies, Expert Syst. Appl. 36 (4) (2009) 7346–7354.

[9] R. Chidamber, F. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493.

[10] C. Couto, P. Pires, M.T. Valente, R. Bigonha, A. Hora, N. Anquetil, Bugmaps-Granger: a tool for causality analysis between source code metrics and bugs, in: Brazilian Conference on Software: Theory and Practice, 2013, pp. 1–6.

[11] M. D'Ambros, M. Lanza, R. Robbes, An extensive comparison of bug prediction approaches, in: 7th IEEE Working Conference on Mining Software Repositories, IEEE, 2010, pp. 31–41.

[12] J. Fehmi, G.G. Yann, H. Sylvie, K. Foutse, Z. Mohammad, Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults, Empir. Softw. Eng. 21 (3) (2016) 896–931.

[13] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, E. Figueiredo, A review-based comparative study of bad smell detection tools, in: 20th International Conference on Evaluation and Assessment in Software Engineering, ACM, 2016, p. 18.

[14] K.A.M. Ferreira, M.A. Bigonha, R.S. Bigonha, L.F.O. Mendes, H.C. Almeida, Identifying thresholds for object-oriented software metrics, J. Syst. Softw. 85 (2012) 244–257.

[15] T.G.S. Filó, M.S. Bigonha, K.M. Ferreira, A catalogue of thresholds for object-oriented software metrics, in: Proc. of International Conference on Advances and Trends in Software Engineering, 2015, pp. 48–55.

[16] F. Fontana, M. Mangiacavalli, D. Pochiero, M. Zanoni, Experimenting refactoring tools to remove code smells, in: Proceedings of the Scientific Workshops on the 16th Int. Conference on Agile Software Development, ACM, 2015, pp. 1–8.

[17] F.A. Fontana, V. Ferme, M. Zanoni, A. Yamashita, Automatic metric thresholds derivation for code smell detection, in: Proceedings of the Sixth International Workshop on Emerging Trends in Software Metrics, IEEE, 2015, pp. 44–53.

[18] M. Fowler, Refactoring: Improving the Design of Existing Code, Pearson Ed., 1999.

[19] T. Graves, A. Karr, J. Marron, H. Siy, Predicting fault incidence using software change history, IEEE Trans. Softw. Eng. 26 (7) (2000) 653–661.

[20] T. Gyimothy, R. Ferenc, I. Siket, Empirical validation of object-oriented metrics on open source software for fault prediction, IEEE Transactions on Software engineering 31 (10) (2005) 897–910.

[21] M.L.C. Januário, K.A.M. Ferreira, Aprimoramento de uma ferramenta de medição de software, Technical Report, CEFET-MG, 2016.

[22] X. Jing, F. Wu, X. Dong, F. Qi, B. Xu, Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning, in: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 496–507, doi:10.1145/2786805.2786813.

[23] X. Jing, F. Wu, X. Dong, B. Xu, An improved sda based defect prediction framework for both within-project and cross-project class-imbalance problems, IEEE Trans. Softw. Eng. 43 (4) (2017) 321–339.

[24] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, J. Liu, Dictionary learning based software defect prediction, in: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, NY, USA, 2014, pp. 414–423, doi:10.1145/2568225.2568320.

[25] S. Kaur, S. Singh, H. Kaur, A quantitative investigation of software metrics threshold values at acceptable risk level, Int. J. Eng. Res.Technol. 2 (2013).

[26] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, (Firs ed.), Springer, 2010.

[27] L. Lavazza, S. Morasca, An empirical evaluation of distribution-based thresholds for internal software measures, in: Proceedings of the 12th International Conference on Predictive Models and Data Analytics in Software Engineering PROMISE, ACM, New York, NY, USA, 2016, pp. 6:1–6:10, doi:10.1145/2972958.2972965.

[28] H. Liu, Z. Ma, W. Shao, Z. Niu, Schedule of bad smell detection and resolution: a new way to save effort, Trans. Softw. Eng. 38 (1) (2012).

[29] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, A. von Staa, On the relevance of code anomalies for identifying architectural degradation symptoms, in: Proceedings of the 16th Europe Conference on Software Maintenance and Reengineering, IEEE, 2012, pp. 277–286.

[30] R. Marinescu, Detection strategies: metrics-based rules for detecting design flaws, in: Software Maintenance., IEEE, 2004, pp. 350–359.

[31] S. Morasca, L. Lavazza, Slope-based fault-proneness thresholds for software engineering measures, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, 2016, pp. 1–10, doi:10.1145/2915970.2915997.

[32] S. Morasca, L. Lavazza, Risk-averse slope-based thresholds: definition and empirical evaluation, Inf. Softw. Technol. 89 (2017) 37–63, doi:10.1016/j.infsof.2017.03.005.

[33] E. Murphy-Hill, A.P. Black, An interactive ambient visualization for code smells, in: 5th international Symposium on Software Visualization, ACM, 2010, pp. 5–14.

[34] M. Nagappan, T. Zimmermann, C. Bird, Diversity in software engineering research, in: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 466–476.

[35] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: Proceedings of the 28th International Conference on Software Engineering, 2006, pp. 452–461.

[36] R. Nascimento, C. Sant'Anna, Investigating the relationship between bad smells and bugs in software systems, in: SBCARS, 2017, pp. 1–10.

[37] H. Nunes, Identificação de bad smells em software a partir de modelos UML, UFMG, 2014 Master's thesis.

[38] P. Oliveira, M.T. Valente, F. Lima, Extracting relative thresholds for source code metrics, in: Software Evolution Week – IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, 2014, pp. 254–263.

[39] J. Padilha, E. Figueiredo, C. Sant'Anna, A. Garcia, Detecting god methods with concern metrics: an exploratory study, in: Proceedings of the 7th Latin-American Workshop on Aspect-Oriented Software Development, 2013, pp. 1–6.

[40] T. Paiva, A. Damasceno, J. Padilha, E. Figueiredo, C. Santanna, Experimental evaluation of code smell detection tools, in: III Workshop on Software Visualization, Evolution, and Maintenance, 2015, pp. 1–8.

[41] M. Riaz, E. Mendes, E. Tempero, A systematic review of software maintainability prediction and metrics, in: Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, 2009, pp. 367–377.

[42] D. Sahin, M. Kessentini, S. Bechikh, K. Deb, Code-smell detection as a bilevel problem, Trans. Softw. Eng.Methodol. 24 (1) (2014) 6.

[43] B. Saida, E.K. El, G. Nishith, R. Shesh, Thresholds for object-oriented measures, in: Proceedings of the 11th International Symposium on Software Reliability Engineering, IEEE, 2000, pp. 24–38.

[44] F. Salfner, M. Lenk, M. Malek, A survey of online failure prediction methods, ACM Comput. Surv. 42 (3) (2010) 10.

[45] R. Shatnawi, W. Li, J. Swain, T. Newman, Finding software metrics threshold values using roc curves, J. Softw. Maint. Evolut. 22 (1) (2010) 1–16.

[46] S. Singh, K. Kahlon, Object oriented software metrics threshold values at quantitative acceptable risk level, CSI Trans. ICT 2 (3) (2014) 191–205.

[47] B.L. Sousa, P. Souza, E. Fernandes, K. Ferreira, M.S. Bigonha, Findsmells: flexible composition of bad smell detection strategies, in: 25th International Conference on Program Comprehension, 2017, pp. 360–363.

[48] P.P. Souza, B.L. Sousa, K. Ferreira, M.S. Bigonha, Applying software metric thresholds for detection of bad smells, in: Proceedings of 11th Brazilian Symposium on Software Components Architecture, and Reuse, ACM, 2017. 10, (in Portuguese)

[49] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, J. Noble, The qualitas corpus: a curated collection of java code for empirical studies, in: Asia Pacific Software Engineering Conference, 2010, pp. 336–345.

[50] R. Terra, L.F. Miranda, M.T. Valente, R. Bigonha, Qualitas.class corpus: a compiled version of the qualitas corpus, Softw. Eng. Notes 38 (5) (2013) 1–4.

[51] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Jdeodorant: identification and removal of type-checking bad smells, in: Software Maintenance and Reengineering., IEEE, 2008, pp. 329–331.

[52] S.A. Vidal, H.C. Vázquez, J.A.D. Pace, C. Marcos, A.F. Garcia, W.N. Oizumi, Jspirit: a flexible tool for the analysis of code smells, in: 34th International Conference of the Chilean Computer Science Society, 2015, pp. 1–6.

[53] G. Vale, D. Albuquerque, E. Figueiredo, A. Garcia, Defining metric thresholds for software product lines: a comparative study, in: Proceedings of the 19th International Conference on Software Product Line, ACM, 2015, pp. 176–185.

[54] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science and Business Media, 2012.

[55] F. Wu, X. Jing, Y. Sun, J. Sun, L. Huang, F. Cui, Y. Sun, Cross-project and within-project semisupervised software defect prediction: a unified approach, IEEE Trans. Reliab. 67 (2) (2018) 581–597.

[56] F. Wu, X.Jing, X. Dong, J. Cao, M. Xu, H. Zhang, S. Ying, B.Xu, Cross-project and within-project semi-supervised software defect prediction problems study using a unified solution, in: IEEE/ACM 39th International Conference on Software Engineering Companion, 2017, pp. 195–197, doi:10.1109/ICSE-C.2017.72.

[57] M. Zhang, N. Baddoo, P. Wernick, T. Hall, Improving the precision of fowler's definitions of bad smells, in: 32nd Annual IEEE Software Engineering Workshop, 2008, pp. 161–166.

[58] M. Zhang, N. Baddoo, P. Wernick, T. Hall, Prioritising refactoring using code bad smells, in: IEEE 4th International Conference on Software Testing, Verification and Validation Workshops, 2011, pp. 458–464.

[59] M. Zhang, T. Hall, N. Baddoo, Code bad smells: a review of current knowledge, J. Softw. Maint. Evolut. 23 (3) (2011) 179–202.

[60] M. Zhang, T. Hall, N. Baddoo, P. Wernick, Do bad smells indicate "trouble" in code? in: DEFECTS, ACM, 2008, pp. 43–44.

[61] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, Vouk, On the value of static analysis for fault detection in software, IEEE Trans. Softw. Eng. 32 (4) (2006) 240–253.