

Exploratory Study of the Impact of Project Domain and Size Category on the Detection of the God Class Design Smell

Khalid Alkharabsheh · Yania Crespo ·
Manuel Fernández-Delgado · José R.
Viqueira · José A. Taboada

Received: date / Accepted: date

Abstract Design smell detection has proven to be an efficient strategy to improve software quality and consequently decrease maintainability expenses. This work explores the influence of information of project context expressed as project domain and size category information, on the automatic detection of the *god class* design smell by machine learning techniques. A set of experiments using eight classifiers to detect *god classes* was conducted on a dataset containing 12,587 classes from 24 Java projects. The results show that classifiers change their behavior when they are used on datasets that differ in these kind of project information. The results of *god class* design smell detection can be improved by feeding machine learning classifiers with this project context information.

Keywords Design Smell Detection · Machine Learning · Software Metrics · Project Context Information · God Class.

1 Introduction

Software quality is an important concern for practitioners in software factories, as well as for academics and researchers. To maintain software quality, the main

K. Alkharabsheh
Department of Software Engineering
Al-Balqa Applied University (BAU)
As-Salt 19117, Jordan.
E-mail: khalidkh@bau.edu.jo

M. Fernández-Delgado, J.R. Viqueira, J.A. Taboada
CíTIUS, Centro Singular de Investigación en Tecnoloxías Intelixentes,
Universidade de Santiago de Compostela, Santiago de Compostela 15782. Spain.
E-mail: {manuel.fernandez.delgado,jrr.viqueira,
joseangel.taboada}@usc.e

Y. Crespo (corresponding author)
Departamento de Informática. Escuela de Ingeniería Informática.
Campus Miguel Delibes, Universidad de Valladolid
Paseo de Belén 15, Valladolid 47011. Spain.
E-mail: yania@infor.uva.es

1 processes that need to be undertaken are identifying bad pieces, modifying code,
2 correcting bugs, adding new functionalities into the software structure, and testing.

3 The poor and bad pieces in a software structure (either in the source code or
4 design) are termed “*design smells*”. *Design smells* do not produce compilation or
5 run-time errors (Alkharabsheh et al., 2018), but they negatively affect software
6 quality factors such as reusability, stability, flexibility and maintainability (Brown
7 et al., 1998). We use the term *design smell*, as (Alkharabsheh et al., 2018) does,
8 because we consider that it integrates different smells which can manifest in the
9 code but are caused by deficiencies in the design. Note that other alternative terms
10 are used in the literature, such as code smells (coined by Kent Beck in Ward’s
11 Wiki¹ as a hint that something has gone wrong somewhere in the code), anti-
12 patterns (Brown et al., 1998), bad smells (Fowler and Beck, 1999) or architectural
13 smells (Azadi et al., 2019), among others.

14 Great interest has been observed in the state of the art with respect to the
15 identification and correction of *design smells* using different approaches and tech-
16 niques (Alkharabsheh et al., 2018; Rasool and Arshad, 2015; Zhang et al., 2011).
17 Many researchers have attempted to tackle this problem by using metric-based,
18 rule-based approaches (Choinzon and Ueda, 2006; Fourati et al., 2011; Marinescu
19 et al., 2005; Moha, 2007; Moha and Guéhéneuc, 2007; Munro, 2005; Shatnawi
20 2015; Tahvildar and Kontogiannis, 2004) and machine learning approaches (Has-
21 saine et al., 2010; Khomh et al., 2011; Kreimer, 2005; Maneerat and Muenchaisri
22 2011). All these studies depend on the definition of the *design smells*, which is
23 mapped into rules manually or automatically; these rules are specified by using
24 a combination of metrics that is directly related to the definition of the *design*
25 *smells*. However, a substantial effort is needed to identify the right threshold value
26 for each metric.

27 For example, the *god class*, which is known in the literature as the *god class*
28 *or the blob* anti-pattern (Brown et al., 1998), *god class disharmony* (Lanza and
29 Marinescu, 2007), and that is similar in definition to the *large class* bad smell
30 (Fowler and Beck, 1999), is characterized by a class that tends to be very large
31 and complex and also has low cohesive functionality. Therefore, the rule that is
32 used to detect the *god class* should include a set of metrics related to class size,
33 complexity and functionality. According to Riel (1996), the *god class* is defined
34 as the class of an object controlling too many objects in the system, which has
35 grown beyond all logic to become *the class that does everything*. Fowler and Beck
36 (1999) described the bad smell *large class* in similar definition, where the class
37 tries to do much tasks. In good object-oriented designs, the logic of the system is
38 uniformly distributed across multiple classes. A *god class* has too many instance
39 variables (attributes), a large number of methods, relations with other classes, i.e.,
40 too much code, which increases the danger of duplicated code.

41 The *god class* is one of the *design smells* which has attracted more attention of
42 the research community (Alkharabsheh et al., 2016a,b; Counsell and Mendes, 2007;
43 Fontana et al., 2012a; Li and Shatnawi, 2007a; Santos et al., 2013; Yamashita et al.,
44 2015). Moreover, it is one of the most frequently detected and hence deserve efforts
45 devoted to improve detection and management of this *design smell*, as revealed
46 in the systematic mapping we conducted (Alkharabsheh et al., 2018). According
47 to (Moha et al., 2010) and (Yamashita and Moonen, 2013), who investigated the

50 ¹ <https://wiki.c2.com/?CodeSmell>

relationships between different types of *design smells*, they found a relationship between the presence of *god class* and of other *design smells*, such as *data class*, *duplicated code*, *feature envy*, *large class* and *god method*. Besides, *god class* is one of the *design smells* that most negatively influence on a broad set of software quality factors such as *maintainability*, *understandability*, *changeability*, *stability*, *performance*, *complexity*, *reusability*, and *evolvability* (Alkharabsheh et al., 2018).

Despite the diversity of approaches, tools and techniques in the literature that could be used to improve the detection of *design smells*, the adoption of appropriate techniques remains a challenge for the software industry. From our review of the state of the art (Alkharabsheh et al., 2018), we observed the following. Firstly, differences in understanding the precise definition of the *design smell* lead to inconsistency: 1) in the results obtained by different software detection tools (Alkharabsheh et al., 2016a) or by human evaluators (Alkharabsheh et al., 2016b); 2) in the real impacts that *design smells* have on a particular software quality factor; and 3) in identifying the most important smells to remove from a software. Secondly, the process of mapping the definition of the *design smell* into accurate and efficient detection algorithms is not exploited adequately due to the ambiguities in the definition of the *design smell* and its mapping to metrics, but also because some adaptation to the context, the project or the organization, may be required.

Most of the research community has not taken into account the impact of project context factors in *design smell* detection (Santos et al., 2018). Our research assumption is that the criteria used by experts to identify *god classes* are influenced by some features of the assessed projects which are not explicitly defined. As an example, larger size or complex classes are more likely to be acceptable (i.e., not considered as *god classes*) in certain sophisticated application **domains**. Similarly, large **size** projects can have the same type of impact in the size and complexity of acceptable classes. The present research investigates whether both project features, domain and size category, have a positive influence on the detection of *god class design smell*. This is further elaborated in subsection 2.2.

On the basis of this assumption, in this paper we investigate whether the detection heuristics of the *design smell* has to be tuned according to the domain of the assessed project and its size classification, which are frequent features when analyzing different types of projects. The main contribution of this study is to test the influence of these features by evaluating the performance of eight machine learning classifiers in the automatic detection of *god classes* with and without project domain and project size category information. We use a large dataset where the *god classes* are labeled by a committee of five *design smell* detection software tools. First we study jointly the influence of project domain and size category, and afterwards we analyze each feature independently. This is followed by an analysis on a dataset in which the classification has been carried out by human experts in order to show whether the results vary.

The rest of this paper is organized as follows. Section 2 presents some related work, particularly Subsection 2.2 deals with the project context information. Next, Section 3 describes the problem to be solved, the Machine Learning techniques (Subsection 3.1), data collection (Subsection 3.2) and methodology (Subsection 3.3). Section 4 discusses the results of the experiments designed in the previous section. After that, a replication of the Step 1 of the methodology, using a different dataset, is described in Section 5 together with a joint analysis

by classifier of all the experiments conducted. Section 6 presents the main threats to their validity. Finally, Section 7 presents our conclusions and the direction we intend to take in future work.

2 Related Work

2.1 On *design smells* detection and Machine Learning techniques

Several studies have focused on *design smell* detection using different strategies, which range from manual to semiautomated and fully automated inspection. The software tools that deal with detecting *design smells* automatically in the software structure include *iPlasma* (Marinescu et al., 2005), a tool that uses a metrics-based approach to detect *design smells* (named disharmonies) at different abstraction levels. In their methodology, more than 80 metrics related to size, complexity, coupling and cohesion can be applied to analyze different software artifacts. The works (Moha and Guéhéneuc, 2007) and (Moha et al., 2010) introduced *DECOR*, a tool that uses a domain specific language to specify *design smells* at a high level of abstraction, where the specification can be defined manually by the practitioners in order to detect *design smells* using software context and vocabulary. In (Tsantalis et al., 2008) *JDeodorant* is presented, an Eclipse plug-in tool that identifies *design smells* in source code and applies a set of refactoring operations to remove them automatically. *JDeodorant* uses several methods and techniques to identify four types of *design smell* (*feature envy*, *type-checking*, *long method*, and *god class*). Some other detection tools such as *Checkstyle* (<http://www.checkstyle.sourceforge.net>) and *FXCop* (<https://msdn.microsoft.com/en-us/library/bb429476.aspx>) are related to bugs, dead code or unused code, and coding style problems, but they do not address higher level *design smells*.

The authors of (Palomba et al., 2015; Rapu et al., 2004; Tufano et al., 2015) employed the changing of software history information for *design smells* detection. In (Palomba et al., 2015), they proposed the *HIST* approach to detect five *design smells* (*feature envy*, *blob*, *divergent change*, *shotgun surgery* and *parallel inheritance*) by analyzing the changes among different software artifacts. This approach was evaluated empirically by two studies and the accuracy was assessed in terms of precision and recall. In (Rapu et al., 2004), the historical information was used to refine the concepts of detection strategy for *god class* and *data class design smells*, by defining a new historical measurement to increase the detection accuracy. In (Tufano et al., 2015), a metric-based methodology is developed to analyze the source code artifacts for identifying different types of *design smells*. In order to investigate when and why *design smells* are introduced in code, this work conducted a large empirical study of the change history of 200 open source projects from different domains to detect five types of *design smells* that include *the blob*, *class data should be private*, *complex class*, *functional decomposition*, and *spaghetti code*. The results conclude that *design smells* are introduced mainly during the maintenance tasks.

Other studies exploited the developers context for improving the *design smells* detection. In (Palomba et al., 2014a), the aim was to ascertain to what extent developers understand *design smells* as problems that should be solved. Moreover, they wanted to check which *design smells* are considered the most harmful. The

1
2 145 results showed that several *design smells* were not acknowledged as problems to
3 146 solve. Nevertheless, *design smells* related to size and complexity such as *god class*
4 147 were always seen as problems. ReCon (Morales et al., 2017) is an example of refactor-
5 148 ing approach related to the developer task context, which uses a metaheuristics
6 149 technique to compute the best order of refactoring operation that influences only
7 150 the code artifacts that are relevant to the developer context.

8 151 On the other hand, several studies used machine learning approaches to detect
9 152 *design smells* compared with metric-based and rule-based. For instance, (Kreimer
10 153 2005) presented a method to detect design flaws by combining object-oriented
11 154 metrics and machine learning techniques. The proposed approach was validated
12 155 using five design flaws (*the blob*, *feature envy*, *long method*, *lazy class* and *delegator*)
13 156 and two software systems (IYC, with 91 classes, and Weka, with 597 classes). More
14 157 recently, (Maneerat and Muenchaisri, 2011) proposed a methodology that uses
15 158 seven machine learning algorithms and seven state-of-the-art datasets consisting
16 159 of 27 design model metrics to predict seven smells (*lazy class*, *feature envy*, *middle*
17 160 *man*, *message chains*, *long method*, *long parameter list* and *switch statement*) using
18 161 software design models.

19 162 The Bayesian detection expert (BDTEX) is a goal question metric-based ap-
20 163 proach that detects anti-patterns using Bayesian belief networks founded on rule-
21 164 based representation (Khomh et al., 2011). This approach was validated using
22 165 three anti-patterns (*the blob*, *functional decomposition*, and *spaghetti code*) and
23 166 two Java projects (Xerces v2.7.0, with 589 classes, and GanttProject v1.10.2, with
24 167 188 classes). SVMDetect (Maiga et al., 2012b) uses the support vector machine
25 168 and object-oriented metrics for each class to detect four anti-patterns (*the blob*,
26 169 *functional decomposition*, *spaghetti code*, and *swiss army knife*) on three open
27 170 source software solutions (ArgoUML v0.19.8, with 1,230 classes, Azureus v2.3.0.6,
28 171 with 1,449 classes, and Xerces v2.7.0, with 513 classes). Afterwards, (Maiga et al.,
29 172 2012a) proposed *SMURF*, an approach to detect anti-patterns also based on the
30 173 support vector machine that takes into account feedback from developers. More
31 174 recently, (Peiris and Hill, 2014) proposed *NiPAD*, a non-intrusive machine learning
32 175 approach for identifying and classifying anti-patterns. Classifiers were trained on
33 176 two datasets of metrics, and then used to predict new metrics. The results showed
34 177 that the approach can detect *one lane bridge* anti-pattern with 0.94 accuracy.

35 178 In (Fontana et al., 2016), a previous work (Fontana et al., 2013b) is extended,
36 179 performing a large-scale comparison of 32 machine learning techniques and 74
37 180 projects to detect four *design smells*: *data class*, *god class*, *feature envy*, and *long*
38 181 *method*. According to the authors, most of the selected classifiers obtained accu-
39 182 racy and F-measures values above 95%, being random forest and J48 the best-
40 183 performing ones. These results show that machine learning approaches can im-
41 184 prove *design smells* detection. Recently, in (Di Nucci et al., 2018), a criticism to
42 185 the current state of the research in ML about *design smells* detection is devel-
43 186 oped, concluding that more work is required regarding to methodology, datasets
44 187 and metric analysis in classes with and without smells. They also analyze the im-
45 188 pact of some dataset properties such as class unbalancing, and of feature selection,
46 189 on the detection of several *design smells* types, using F-measure as performance
47 190 measurement.

48 191 A systematic literature review and meta-analysis of machine learning tech-
49 192 niques for smell detection was conducted in (Azeem et al., 2019). This study con-

cludes, among other aspects, that other sources of information for the classification of smelly instances should be investigated.

The main difference between our approach and previous works that employed machine learning is that they only included numerical information (metrics) in the dataset. Also, they used just two or three projects in the validation process, conducting experiments with less than eight machine learning technique, with the exception of (Fontana et al., 2016) and (Di Nucci et al., 2018), who used 74 projects and 32 techniques. In contrast, the current work extends numerical information (metrics) with project context information (its domain and size category expressed in nominal scales) and our objective is to evaluate its influence on the *god class* detection, considering 12,587 classes from 24 open source Java projects, and eight machine learning classifiers arising from different families. Another important difference is in the results evaluation: the current paper uses the Cohen Kappa and Matthews correlation coefficient (MCC) as performance measurements. Studies as (Bekkar. et al., 2013) shown that problems with unbalanced data should be better evaluated measuring some indicators such as MCC, Cohen Kappa and Receiving Operating Characteristic (ROC) curve (Powers, 2011). When assessing a project for *god class design smell* detection, it is normal that data are imbalanced because just a small ratio of the project classes can be considered as *god classes*.

Recently, in (Pecorelli et al., 2019) a comparison between heuristic-based detection and machine learning detection techniques is conducted. The experiment analyzes five machine learning algorithms in front of an heuristic-based tool (DECOR). The study was conducted with five different smells. The authors use as performance indicators precision, recall, F-Measure and MCC. A dataset with different versions of 13 projects was used. The results shows that the best classifier of the five used was Naive Bayes, and that the heuristic approach in DECOR still perform better, yet have low performance. The problem with unbalance datasets in *design smells* detection is also highlighted.

As a consequence of the last conclusion, in (Pecorelli et al., 2020) an empirical study comparing the performance of five data-balancing techniques for code smell detection with respect to a no-balancing baseline was conducted. The data-balancing techniques included those which perform training only on the minority class such as Cost-Sensitive Classifier, and One-Class Classifier, resampling techniques such as Oversampling and Undersampling, and Creating synthetic instances (SMOTE). The results obtained shows that SMOTE is the best data balancing technique among the other five. However, authors state that if the dataset is exceptionally imbalanced, this technique fails because, although SMOTE allows the model to be more accurate, in many cases, it is not applicable because of the few instances belonging to the minority class. The authors concluded, on one hand, that existing data balancing techniques are inadequate for code smell detection and, on the other hand, that their results indicate that structural metrics alone are not adequate for code smell detection, which is in the line of the ground assumption that motivates our work.

Finally, a multi-label classification (MLC) machine learning technique is analyzed in (Guggulothu and Moiz, 2020) for two method-level code smells: Feature Envy and Long Method. The authors use the smell correlation and experimented with three different MLC techniques, analyzing with and without the usage of the correlation between smells.

241 2.2 Project context information

242 In this subsection the software context on the identification of design smells (code
243 smells, bad smells, anti-patterns, . . .) is explored in depth.

244 The antecedents of using context information to detect smells are found in what
245 it is known as “relative metrics thresholds”. In (Crespo et al., 2006) it is stated that
246 most of the tools for metric-based smells detection use generic values for thresholds.
247 Authors dealt with the question of whether using generic vs. relative thresholds
248 affects the detection of bad smells and argue that product domain and size could
249 also affect the results. In (Alves et al., 2010) a method that determines metric
250 thresholds using measurements in a benchmark of software systems is designed.
251 Machine learning and data mining techniques were utilized in (Herbold et al.,
252 2011) to define a data-driven methodology for the calculation of thresholds for a
253 metric set using Rectangle Learning. Authors stated that an important aspect of
254 thresholds for metrics is that they are often dependent on the properties of the
255 project environment. In order to evaluate results they used a case study assessed
256 by measuring Matthews Correlation Coefficient (MCC) and F-score (F-measure).
257 Moreover, in (Fontana et al., 2015), a benchmark-based methodology for deriving
258 metrics’ thresholds for code smell detection is presented.

259 In the conclusions of the study presented at (Fontana et al., 2012b), the authors
260 stated that to improve the smells’ detection techniques, information related to the
261 domain of the analyzed systems should be taken into account. A large-scale study
262 that investigated the relationships between the presence of smells and quality-
263 related metrics computed over Java Mobile apps in different application domains
264 was conducted in (Linares-Vásquez et al., 2014). The authors found that some
265 smells are commonly present in all the domains while others are most prevalent
266 in certain domains. DECOR was used as anti-patterns & smells detection tool.
267 Although the referred work is not intended to improve detection by adapting to the
268 domain as part of the project context, the authors state, on one side, that “domain
269 matters” and, on the other side, that “metrics are not enough”. This is the same
270 statement that can be found in (Simons et al., 2015) regarding a very close topic
271 which is refactoring to improve quality. Through qualitative analysis of a survey
272 of professionals, answered by 50 engineers, authors find that a simple static view
273 of software is insufficient to assess software quality, and that software quality is
274 dependent on factors that are not amenable to measurement via metrics. As part
275 of this qualitative analysis, authors found that “The Problem Domain Matters”
276 and also that “Qualities Have Meaning only in a Given Context”.

277 Based on (Crespo et al., 2006), in (Liu et al., 2016) a method with genetic
278 algorithms to dynamically adapt thresholds in smells detection was designed and
279 tested. This method captures the developer/project context through incorporat-
280 ing engineers feedback in the process. Engineers could adapt metrics’ based smell
281 detection tools with the so calculated thresholds. The method includes collecting
282 feedback automatically from the engineers who manually check and resolve smells.
283 Hence, it collects information about which smells have been confirmed manu-
284 ally, and which smells have been denied. Five open-source projects were used in
285 evaluation. Authors assessed results using precision and recall. Manual applica-
286 tion of detection algorithms was achieved by three engineers and JDeodorant was
287 used for metric collection. With this approach, we can not infer what are the im-
288 portant context factors that influence in improving smell detection. In contrast,

(López Nozal, 2012) used J48 algorithm in order to obtain decisions trees and find that domain and project size can influence on improving detection because rules regarding this data appear in the obtained decisions trees. It inspired us to conduct this exploratory study to assess the positive influence of introducing project context information, mainly domain and size category, in machine learning based automatic *god class* detection.

Moreover, Chapter 1 of (Mistik et al., 2015) states that metric's thresholds may vary considerably depending on software size, cost, nature of team, etc. In (Mori et al., 2018) authors observed that software domain and size are relevant factors to be considered when building benchmarks for metric's threshold derivation. Recently, this work was continued in (Vale et al., 2019), calculating thresholds for a benchmark with three different methods as a comparison with the Vale's threshold calculation method. Authors use an SPL called MobileMedia. They tune smell detection such as *god class* and lazy class with the so calculated thresholds, and observe that using the Vale's thresholds improves the detection of those smells in terms of precision and recall when applying Lanza & Marinescu strategy for detecting *god class* and lazy class manually with the different thresholds in the same line as (Liu et al., 2016).

3 Study Design

This section is organized as follows, first of all, in Subsection 3.1 we describe the machine learning classification techniques used in this study to automatically detect *god classes*. After that, we describe the data collection in Subsection 3.2. Finally, in Subsection 3.3 the methodology applied is explained.

The scope of the study is limited to *god class* in first place because of the nature of the smell. The hypotheses on the influence of the domain and size of the project on the detection of *god class* can be intuitively explained before conducting the experiments. In second place, the design of the methodology uses combinations of labeling by the five tools, as is detailed in next section: only one tool detects the *god class*, two of the tools detect the *god class*, three, four or all five detect the *god class*. This implies that the five tools must be able to analyse the same smell. In the systematic mapping study we conducted on smells detection (Alkharabsheh et al., 2018), it is shown that it is not usual to find that several tools detect a common subset of smells. Human validation on the cases of the God Classes detected simultaneously by four of the tools and the five tools to check the false positive reduction are performed. In the first case we detect 8 false positives among the 79 god classes and, in the second case only one false positive was detected, particularly strange, among the 24 god classes detected. This is explained in Section 4 when describing results of step 4 of the methodology (page 24). At the end, a replication experiment is conducted using a manually classified dataset which is well known in the literature (Pecorelli et al., 2019, 2020).

3.1 Machine Learning Techniques

We selected several classification techniques that are widely applied for smell detection (Khomh et al., 2011; Kreimer, 2005; Maneerat and Muenchaisri, 2011; Fontana

et al., 2013b). These techniques use information about the code of the classes (which in the literature is called an input pattern, see below) to decide whether it is a *god class*, i.e., there are only two possible labels: “*god class*” and “*not god class*”. We use several techniques in order to avoid that the conclusions of our study may be biased by the specific classifier. These classifiers represent different approaches such as Bayesian networks, support vector machines, rule learners and decision trees. Implementations of the chosen techniques are available in the well-known data mining software Weka version 3.7 (<http://www.cs.waikato.ac.nz/ml/weka>). Most classifiers have one or several tunable hyper-parameters, which usually have strong influence on their performance. The set of values used for the hyper-parameter tuning is extracted from the classifier documentation (Witten et al., 2016). The following list briefly describes each classifier and its hyper-parameter tuning.

1. LibSVM is an integrated library for support vector classification, regression and distribution estimation. The library is available with different software interfaces and extensions, such as Weka, Matlab, C, Java and Python. In our experiments, the classifier used is the Gaussian kernel SVM, whose high performance has been widely tested in the literature. The tunable hyper-parameters are regularization (C) and the inverse of the squared Gaussian spread (γ), tuned by selecting 20 values in the ranges 2^{2i+1} , with $-3 \leq i \leq 7$, and 2^{2i+1} , with $-8 \leq i \leq 1$, respectively.
2. IBK is the instance-based k -neighbor classifier, which classifies each input pattern to the label of the majority of the k closest neighbors for training. The LinearNNSearch is the selected search algorithm with IBK. The number of nearest neighbors (k) is tuned using eight odd values between 1 and 41.
3. J48 is the Weka implementation of the C4.5 decision tree algorithm, which is also a popular classification method. It can produce rules that are understandable and intelligible for classifying new instances. The confidence interval (C) is tuned using the values 0.1, 0.25 and 0.5.
4. JRip is a learning algorithm based on association rules with reduced error pruning (REP). It produces logical rules for classification that are easy to understand and that can be converted into source code. It has two tunable hyper-parameters: number of folds for REP ($F \geq 3$, 4 values) and number of optimization runs ($O \geq 2$, 3 values).
5. SMO, uses the sequential minimal optimization algorithm to solve the quadratic programming problem which appears through the training of a support vector machine. It uses heuristics to partition the training dataset into smaller problems that can be solved analytically. In the experiment, the SMO used a square polynomial kernel given by $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T \mathbf{y} + 1)^2$. The complexity hyper-parameter (C) is tuned by selecting 20 values in the set 2^{2i+1} , ($-3 \leq i \leq 7$).
6. NB is the Naive Bayes based on the Bayesian theorem with strong assumptions on the independent features. It belongs to the family of probabilistic classifiers and is suitable for large-size input data. The NB has no tunable hyper-parameter, and we use supervised discretization (D parameter).
7. RC (Random Committee) is an ensemble of random base classifiers, trained using different random seeds on the same data whose output is the average

prediction of the base classifiers, in this case random classification trees. The number of iterations (I) is tuned with five values ($I \geq 10$).

8. RF is the well-known random forest classifier, an ensemble (“forest”) of random decision trees, where each tree has a particular subset of input features. It classifies each pattern by selecting the tree that contains the majority of classifications among all the existing trees. The number of iterations (I) is tuned with four values ($I \geq 100$).

3.2 Data Collection

As mentioned above, we built a dataset composed by 24 open source Java projects containing 12,587 classes and 1,852 KLOC, which were obtained from the SourceForge source code repository. Only projects written in Java were selected because it is the only programming language which is common to all the detection tools used. Furthermore, Java is one of the most common mainstream object-oriented languages, particularly in the context of open source software and *design smell* detection tools and it allows to evaluate many metrics. We also selected projects belonging to different domains and sizes in order to allow us complete the study. Finally, only projects in beta, mature or production/stable status were selected to discard prototypes or experimental software. Specifically, we obtained a set of projects with a long-life cycle (before 2013) with a history of several versions. Given the large number of available projects meeting these criteria in the SourceForge repository, we randomly selected 24 projects, provided that projects of all the domains and sizes are included. We selected the SourceForge repository because it is one of the most widely known, it is used in the context of open source software and it supplies useful metadata for projects.

The classification of a project into a specific domain is not straightforward because there is no general schema available that is universally agreed. Therefore, we followed the classification proposed in (Tempero et al., 2010) in which the projects are distributed into four main domain categories (Table 1), each one containing a set of different sub-domains which are not used in this study. With respect to project size, the categories refer to the TLOC (total lines of code) written for the whole project. This methodology is the most commonly used in software sizing. In the dataset, the project size is divided into five categories from small to large, similarly to the study (Fontana et al., 2013a) of project classification. The categories are defined as follows: S (small) for projects where $TLOC < 5,000$; $5,000 \leq SM$ (small-medium) $< 15,000$; $15,000 \leq M$ (medium) $< 40,000$; $40,000 \leq ML$ (medium-large) $< 100,000$; $100,000 \leq L$ (large) $< 500,000$. This is an ordinal scale, but we are going to treat it as nominal.

In order to classify the 24 projects into the 4 domains, we perform an independent survey with four of the five authors. This first round included a survey question for each project with the url of the project in SourceForge and asking for a rapid answer. After first round, a Kappa inter-rated agreement of 24 subjects and 4 raters was calculated 0.441 as moderate agreement with $p - value = 0$. An analysis of the survey’s answers shows that 9 of the projects have 4/4 coincidence and other 7 have 3/4 coincidence. The rest of the projects have 2, 1, 1 but one of the answers was "Can’t classify". The four authors open a discussion about "Can’t classify" answer that shows that some authors knows previously these projects but

1
2 425 other authors don't. Just visiting the SourceForge page quickly was not enough.
3 426 Hence, the second round starts with an open discussion and sharing more infor-
4 427 mation on the projects. The agreement was achieved and the fifth author review,
5 428 check and agree.

7
8 **Table 1** Domain categories and sub-domains.

Category	Sub-domain
Software Development (Dev)	Parsers/Generators/Makers, Testing, Integrated Development Software (IDE), Software Development Kit (SDK)
Application Software (App)	Word Processor, Web Browser, Accounting, Graphic, Player
Diagram Generator/Data Visualization (Vis)	Graphical User Interface (GUI) Design Tool
Client Server Software (Cli)	Database, Application Server, Middleware, Content Management System (CMS)

17
18
19
20 **Table 2** Characteristics of the projects in the dataset. (App: Application Software; Dev: Soft-
21 ware Development; Vis: Diagram Generator/Data Visualization; Cli: Client Server Software;
22 L: Large; ML: Medium-Large; M: Medium; SM: Small-Medium; S: Small.)

Project name	Domain	Size Category (TLOC)	NOC
jAudio-1.0.4	App	L (117615)	416
JDistlib-0.3.8	App	M (32081)	78
Freemind-1.0.1	Vis	L (106396)	782
JCLEC-4-base	Dev	M (37575)	311
JasperReports-4.7.1	Dev	L (350690)	1797
Java graphplan-1.0	Dev	SM (1049)	50
Squirrel SQL Client-3.7.1	Cli	ML (71626)	1138
Mpxj-4.7	App	L (261971)	553
KeyStore Explorer-5.1	Vis	ML (83144)	384
Apeiron-2.92	App	SM (8908)	62
DigiExtractor-2.5.2	App	M (15668)	80
FullSync-0.10.2	App	M (24323)	169
Angry IP Scanner-3.0	App	M (19965)	270
OmegaT-3.1.8	App	L (121909)	716
Plugfy-0.6	Dev	S (2337)	28
Lucene-3.0.0	App	ML (81611)	606
Matte-1.7	App	ML (52067)	603
Ganttproject-2.0.10	App	ML (66540)	621
sMeta-1.0.3	App	M (30843)	222
JFreechart-1.0.X	App	L (206559)	499
xena-6.1.0	Dev	ML (61526)	1975
JHotDraw-5.2	App	M (17807)	151
pmd-4.3.x	Dev	ML (82885)	800
checkstyle-6.2.0	Dev	ML (41104)	277

42
43 Table 2 provides, for each project, its name and version. This table also char-
44 430 characterizes the projects in terms of domain and size category. The TLOC is included
45 431 because it is used to obtain size category, while the number of classes (NOC) is
46 432 included for further comparison with the number of *god classes*.

47 433 In order to analyze the chosen projects, we carefully selected a source code
48 434 analyzer tool that matched the following criteria: open source, measures Java
49 435 source code and includes a high number of object-oriented metrics. The tool that
50 436 best met these criteria and suited the needs of this study was *RefactorIT* version

2.7 (<https://sourceforge.net/projects/refactorit/>). It uses semantic rules and produces a set of 30 metrics at the levels of project, package, class and method, but in our case we exclude the 14 method metrics because they are not relevant for our study. RefactorIT offers great functionality in terms of the management of the metrics profile and user interface. Also, it supplies a catalog of refactoring operations that assists in the maintenance process. *RefactorIT* is available in two forms: as an integrated plug-in of Eclipse or as a standalone desktop tool.

The dataset requires a particular formalization so that it can be input into the machine learning classifiers. Each class in the dataset is represented by a row of 19 variables (X_1 to X_{19}), see Table 3, widely used in the literature (Khomh et al., 2011; Kreimer, 2005; Maneerat and Muenchaisri, 2011) for *design smell* detection. Variables X_1 to X_{16} involve numerical attributes which represent software metrics and include project ($X_1 \dots X_5$), package ($X_6 \dots X_9$) and class ($X_{10} \dots X_{16}$). Some of these metrics are particularly related to *god class* characteristics such lack of cohesion (LCOM), number of attributes (NOA), and complexity (weighted methods per class, WMC).

Table 3 Definition of variables.

Variable	Metric	Definition	Level
X_1	TLOC	Total Lines of Code	Project
X_2	NCLOC	Non-Comment Lines of Code	Project
X_3	CLOC	Comment Lines of Code	Project
X_4	EXEC	Executable Statements	Project
X_5	DC	Density of Comments	Project
X_6	NOT	Number of Types	Package
X_7	NOTa	Number of Abstract Types	Package
X_8	NOTc	Number of Concrete Types	Package
X_9	NOTe	Number of Exported Types	Package
X_{10}	RFC	Response for Class	Class
X_{11}	WMC	Weighted Methods per Class	Class
X_{12}	DIT	Depth in Tree	Class
X_{13}	NOC	Number of Children in Tree	Class
X_{14}	DIP	Dependency Inversion Principle	Class
X_{15}	LCOM	Lack of Cohesion of Methods	Class
X_{16}	NOA	Number of Attributes	Class
Project level information (nominal scale)			
Variable	Information	Value	
X_{17}	Project domain	App, Dev, Cli, Vis	
X_{18}	Project size category	L, ML, M, MS, S	
Smell detection (binary)			
Variable	Design Smell	According to	
X_{19}	God Class	PMD, iPlasma, JDeodorant, Décor, Together	

Variables X_{17} and X_{18} represent the two types of project context information (domain, size category) as factors. Finally, variable X_{19} represents the class label (*god class* or not) as a binary scale (true or false). Its value should be defined manually by a team of experts. However, it would be very costly, so we used a combination of five *design smell* detection tools which are very popular and cited in the literature according to the results of our systematic mapping study (Alkharabsheh et al., 2018), commonly employed in *god class* detection (Khomh et al., 2011; Lanza and Marinescu, 2007; Fontana et al., 2012a; Yamashita et al., 2015; Maiga et al., 2012b), which offer a feasible and repeatable way to analyze our dataset. The five tools are *iPlasma* (Marinescu et al., 2005), *DECOR* (Moha, 2007), *JDeodorant* (Tsantalis et al., 2008), PMD (Copeland, 2005) and Together

(Borland, 2008). PMD and iPlasma use the same strategy, which is a metric-based strategy. Moreover, both tools use the same three metrics defined in Lanza & Marinescu (Lanza and Marinescu, 2007) strategy but with different thresholds. These three metrics are Weighted Methods per Class (WMC), Access to Foreign Data (ATFD), Tight Class Cohesion (TCC). The general strategy is defined as:

```

469 WMC >= WMC_VERY_HIGH &&
470     ATFD > FEW_ATFD &&
471     TCC < TCC_VERY_LOW

```

PMD considers, according to the coded rule obtained from Github (<https://github.com/pmd/pmd/blob/master/pmd-java/src/main/java/net/sourceforge/pmd/lang/java/rule/design/GodClassRule.java>):

```

475 WMC_VERY_HIGH as 47,
476 FEW_ATFD as 5, and
477 TCC_VERY_LOW as 1/3

```

iPlasma considers, according to (Mihancea and Marinescu, 2005):

```

479 WMC_VERY_HIGH as 20,
480 FEW_ATFD as 4, and
481 TCC_VERY_LOW as 1/3

```

JDeodorant, according to (Fokaefs et al., 2009), uses a hierarchical agglomerative clustering algorithm on each class, obtaining clusters with attributes and methods of a class. This method could be named as feature (attribute and methods) clustering analysis to find opportunities for “Extract class” refactoring.

DECOR is rule based, as a mixture of semantic rules, metrics based rules, structural rules and association rules. DECOR generates the detection code automatically from a definition of the anti-pattern expressed in a domain specific language (DSL) as shown in the rule card below, according to (Moha et al., 2010; Palomba et al., 2014b). The metric rules deal with Number of Methods Declared (NMD), Number of Attributes Declared (NAD) and LCOM5 (a version of Lack of Cohesion of Methods proposed by Henderson-Sellers). Compared with iPlasma and PMD, it is based in different types of rules, not just metrics and it not even uses the same metrics set.

```

495 RULE_CARD: Blob {
496     RULE: Blob {ASSOC: associated FROM: mainClass ONE
497                 TO: DataClass MANY}
498     RULE: mainClass {UNION
499                     LargeClassLowCohesion
500                     ControllerClass}
501     RULE: LargeClassLowCohesion {UNION
502                                     LargeClass
503                                     LowCohesion}
504     RULE: LargeClass {(METRIC: NMD+NAD, VERY_HIGH, 20)}
505     RULE: LowCohesion {(METRIC: LCOM5, VERY_HIGH, 20)}
506     RULE: ControllerClass {UNION
507                             (SEMANTIC: METHODNAME,
508                              {Process, Control, Command,
509                               Manage, Drive, System}),
510                             (SEMANTIC: CLASSNAME,
511                              {Process, Control, Command,
512                               Manage, Drive, System})}
513     RULE: DataClass {(STRUCT: METHOD_ACCESOR, 90)}
514 };

```

Together ControlCenter (Borland Together 12.6) calculates 55 quality assurance metrics for use in analyzing Java and C++ source code (Gronback, 2003). It includes Chidamber and Kemerer (CK) suite and it is also inspired in Lanza & Marinescu (LM). According to different papers in literature, Together's *god class* detection (as shown manually in (Gronback, 2003)) is metric-based using a combination of different metrics, including LM and CK suites but, to the best of our knowledge (see for example (Li and Shatnawi, 2007b)), the precise strategy and thresholds used are not published.

Note that in our data collection the *god class* labeling and the class metrics are independent, because the former is defined by these five detection tools as described before, while the later are computed using RefactorIT. On the other hand, there is a lack of agreement on the detection (Paiva et al., 2017; Fernandes et al., 2016) and in the values of the metrics delivered by the different tools. This suggests that, although some of the software detection tools might use code metrics similar to the ones used by us, rarely are exactly the same metrics, calculated the same and used in the same way, so the probability of biasing in *god class* labeling is very low.

All the tools support Java projects and the input data are the source code, while the output is text in different formats (CSV, TXT or XML). Also, most of the tools focus only on smell detection, except *JDeodorant*, which includes refactoring operations that are performed after the *design smells* have been identified. Nearly all the tools have a GUI except *PMD*, which only provides a CLI (command line interface). Some tools can be executed in different environments such as IDE plugins or standalone and generate metric reports. Table 4 summarizes the most important characteristics of each detection tool. In our experiment these five detection tools are used, each of which must be operated in its way, some adapting the project to be analyzed by the tool, obtaining and processing the result manually by the authors. The aspect named CLI value "No" (in Table 4) describes that the tool should be used throughout the graphical interface. The full dataset used in this study is available in XLS and CSV formats from <https://citius.usc.es/investigacion/datasets/project-nominal-information>

Table 4 Characteristics of selected design smell detection tools.

Tool	DECOR	JDeodorant	iPlasma	PMD	Together
Version	1.0	5.0.13	6.1	5.3.2	12.6.0
Type	Open source	Open source	Open source	Open source	Commercial
Strategy	Rule-based (mix)	Clustering analysis	Metrics-based	Metrics-based	Metrics-based
Language	Java	Java	Java, C++	Java, C, C++	Java, C++, C#
Refactoring	No	Yes	No	No	No
Environment	Standalone	Eclipse Plug-in	Standalone	Eclipse Plug-in, Standalone	Eclipse Plug-in
Metrics Report	No	No	Yes	No	Yes
Inputdata	Source code	Source code	Source code	Source code	Source code, UML
Outputdata	Textual	Textual	Textual, Visual	Textual	Textual
Commandline	No	No	No	Yes	No

Each single tool is not perfect, and specifically it is widely known in the literature that they are prone to detect false *god classes*. Additionally, each detection tool usually performs better on its own developing context, defined by the types of project used, experience of the experts who designed the tool, etc. However, combining several detection tools in a committee with an adequate fusion policy for the outputs of the tools may enhance the detection performance. Specifically, by requiring less tools to detect a *god class*, the undetected *god class* rate is expected to reduce. Conversely, by requiring more tools to detect a *god class*, false detection rate is expected to reduce, specially considering the low agreement among tools. In fact, we performed a manual analysis of cases labeled as *god class*, with extreme policies of 4 and more tools and 5 tools coinciding in detection. In the first case, we found 10% of false positives (8 cases out of 79) while, in the second, 4% (only one case, particularly strange, out of 24). These values are low compared to those reported for these tools in other studies (Moha and Guéhéneuc, 2007; Fontana et al., 2011). We develop several experiments using different policies to combine the *god class* detection tools (e.g., a logical AND/OR of the five tools). These experiments also evaluate whether the influence on the *god class* detection of the information concerning the project context is consistent throughout the different policies.

Figure 1 shows the distribution of *god classes* detected over the dataset (24 projects) by the selected detection tools. The bubbles in the figure represent the number of *god classes* detected in each project by each tool. The two upper bubble rows represent the “Logical AND” of all the tools, i.e., the number of *god class* identified simultaneously by the 5 tools, and the “Logical OR” of all tools, i.e., the *god classes* identified by some tool, in other words, by one tool and more. The largest number of *god classes* was detected by *JDeodorant*, which is an open source tool, while the lowest number was detected by *Together*, which is a commercial tool. The majority of *god classes* were detected in projects that mostly belonged to the App domain and/or had a L size category, such as *OmegaT-3.1.8*, *Lucene-3.0.0*, *GanttProject-2.0.10*, *Xena-6.1.0*, *Jasperreport-4.7.1*, *JAUDIO-1.0.4* and *JFreeChart-1.0.x*. For the same project, different *god class* detection results were obtained by different tools. Studying the agreement between the five selected detection tools, a Fleiss’ Kappa test provides a Kappa value of 0.202, which means poor agreement among the five tools, with a p -value of 0, which means that the differences between the five tools are statistically significant. Therefore, there exists a lack of agreement among the tools, which may be caused by the variety of techniques, algorithms and metric thresholds used by the tools.

From Table 5, it can be seen that the *god classes* are distributed among different categories of the investigated project context information. Since well-designed projects are expected to have zero or few *god classes*, in our dataset this label is low-populated, so the classification problem is unbalanced. For example, the S size category only includes one *god class*, because small-size projects should not include more than one or two *god classes*. In general, there exists a positive relation between the number of classes and the number of detected *god classes* in the same category (domain or project size), although there are some exceptions. For example, the ML size category has 6,404 classes and 758 *god classes*, while the L category that has 4,762 classes but 967 *god classes*. This behavior also happens considering projects instead of classes. The row labeled **Correlation** in Table 5 reports the correlations between the number of total classes and the number of *god*

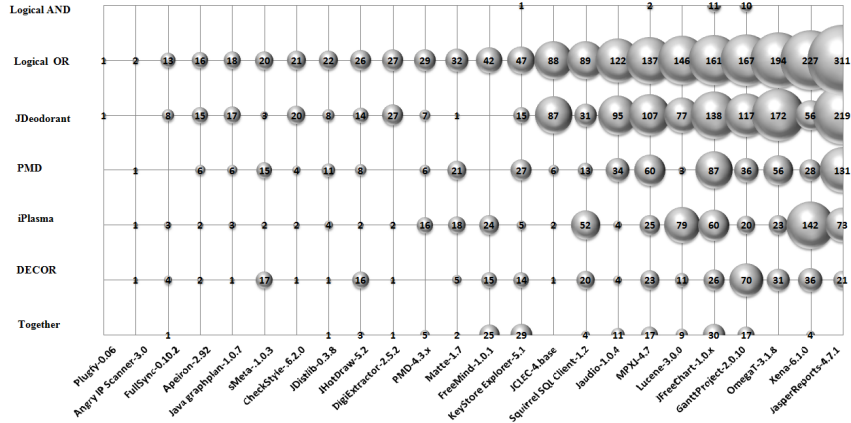


Fig. 1 God class distribution over the dataset using the selected tools.

Table 5 Distribution of *god classes* among different nominal categories (N.Cat.) requiring one and more tools, two and more tools, three and more tools, four and more tools and five tools. Correlations between the number of *god classes* and the number of classes over the domain and size categories are also reported. The last line reports the number of projects, classes, *god classes* and *god class* percentages of the whole dataset for each number of tools.

N.Cat.	Project	Class	#God Classes				
			≥1-tool (%)	≥2-tools (%)	≥3-tools (%)	≥4-tools (%)	5-tools (%)
Dev	7	5238	695 (13.3)	155 (2.9)	42 (0.8)	3 (0.1)	0 (0)
App	14	5046	1085 (21.5)	321 (6.4)	171 (3.4)	70 (1.4)	23 (0.5)
Vis	2	1165	89 (7.6)	43 (3.7)	18 (1.5)	3 (0.3)	1 (0.1)
Cli	1	1138	89 (0.8)	20 (1.8)	8 (0.7)	3 (0.3)	0 (0)
Correlation			0.93	0.85	0.69	0.54	0.53
L	6	4762	967 (20.3)	326 (6.8)	158 (3.3)	48 (1)	13 (0.3)
ML	8	6404	758 (11.8)	165 (2.6)	62 (1)	27 (0.4)	11 (0.2)
M	7	1281	198 (15.5)	36 (2.8)	15 (1.2)	2 (0.2)	0 (0)
SM	2	112	34 (30.4)	12 (10.7)	4 (3.6)	2 (1.8)	0 (0)
S	1	28	1 (3.6)	0 (0)	0 (0)	0 (0)	0 (0)
Correlation			0.93	0.80	0.74	0.84	0.94
Whole dataset	24	12,587	1,958 (15.5)	539 (4.3)	239 (1.9)	79 (0.6)	24 (0.2)

classes for each number of tools: their high positive values show that the latter generally raises with the former. The unbalance between *god classes* and non-*god classes* is also clear in this table (the *god class* percentages never overcome 30.4%, 10.7%, 3.6%, 1.8% and 0.5% for each number of tools used simultaneously to detect the smell). The last row of Table 5 reports the total number of projects, classes and *god classes* and the percentage of *god classes*, for each combination policy (≥ 1 tool, ≥ 2 tool and so on). These percentages never overcome 16%, and they decrease very fast with the number of tools required to label a class as *god class*, increasing very much the classification unbalance.

3.3 Methodology

In this section we describe the experimental methodology, based on the use of several supervised classifiers to predict the presence of *god class design smell*. This methodology includes several steps which are related to the following **research question (RQ)**:

RQ: Does the project context information, specifically project domain and size categories, influence on the detection of the *god class design smell*?

Firstly, we identify *god class design smell* using a detection tool (*DECOR*, *JDeodorant*, *iPlasma*, *PMD*, and *Together*), listed in the section 3.2. Our problem is modeled as a two-class classification problem: label 0 corresponds to “normal class” and label 1 corresponds to “*god class*”. We use the classifiers listed on the section 3.1 to learn this classification problem. In order to build each classifier, the dataset is randomly splitted into a training set (60%), a validation set (20%) and a testing set (20%). Specifically, five randomly splitted groups of training/validation/test sets are generated (keeping the relative populations of both classes in each set), and the classifier performance is averaged over the five trials. Note that, since different policies of combining detection tools are considered, each policy creates a different *god class* labeling, and thus a different dataset. In each trial, the training set is used to train the classifiers and the validation set is used to select appropriate values for their tunable hyper-parameters (e.g., the spread of Gaussian kernel for LibSVM), listed in section 3.1. The hyper-parameter values are selected to provide the best average classification performance over the 5 validation sets. Finally, the classifier, trained on the 5 training sets using the selected values for the tunable hyper-parameters, runs on the 5 test sets, and the final performance is the average over these test sets.

The classification performance is measured by the Cohen Kappa (Blackman and Koval, 2000) statistic, in %, which measures the agreement between the true label and the one predicted by the classifier discarding the agreement by chance (e.g., caused by class unbalance, when all the patterns are classified in the majority class). This property has led us to select kappa over other statistics because the information in our problem is strongly unbalanced. The interpretation of Kappa values, denoted by K , which range from -100% to 100% , is as follows: excellent ($80\% < K < 100\%$), good ($60\% < K < 80\%$), moderate ($40\% < K < 60\%$), fair ($20\% < K < 40\%$) and poor ($K < 20\%$). According to this interpretation, in our study, we consider the classifier we built is good when the Kappa value is greater than 40%. The value of K is calculated as:

$$K = 100 \frac{a - e}{s - e}, \quad a = TN + TP, \quad s = TN + FP + FN + TP \quad (1)$$

$$e = \frac{(TN + FP)(TN + FN) + (FN + TP)(FP + TP)}{s} \quad (2)$$

where TN , FP , FN and TP are the number of true negatives, false positives, false negatives and true positives, respectively, and positives correspond to *God Class*. Note that K is always equal or lower than accuracy. The Matthews correlation coefficient (MCC, also in %) is another performance measure of the quality of binary classifications that is used to evaluate the classifiers (Matthews, 1975). The

MCC has been evaluated in all the experiments, but for brevity we report mainly the Kappa values, although both measurements lead to the same conclusions.

In our experiments, we use the Wilcoxon rank sum test (Hollander et al., 2013) to evaluate the difference between the Kappa (resp. MCC) values achieved by the eight classifiers in two cases, e.g. with and without project context information. The null hypothesis of this test is that Kappa (resp. MCC) values in both cases come from distributions with equal mean. When the p -value is below (resp. above) 0.05, this hypothesis can be rejected (resp. cannot be accepted), so the difference between cases is (resp. is not) statistically significant. The test also allows hypothesis such as “one distribution is greater than or equal to the other”, but the procedure for rejecting the hypothesis is the same. This test is evaluated in R by the `wilcoxsign_test` function of the `coin` package which also provides a Z -factor that divided by \sqrt{N} , where N is the number of pairs of data, provides the effect size (notated by r) in whose interpretation we use the criteria widely accepted in the literature (Cohen, 2013): small effect size means $0.1 \leq r < 0.3$; moderate is $0.3 \leq r < 0.5$; and large is $0.5 \leq r$. These performance measures and this statistical test are widely used in the literature for similar purposes (Maneerat and Muenchaisri, 2011; Hall et al., 2011; Fontana et al., 2012a, 2013b; Di Nucci et al., 2018).

The current study is organized in six steps (Figure 2). The first four steps are applied on the whole dataset, while the last two divide the dataset in different project domains (step 5) and size categories (step 6).

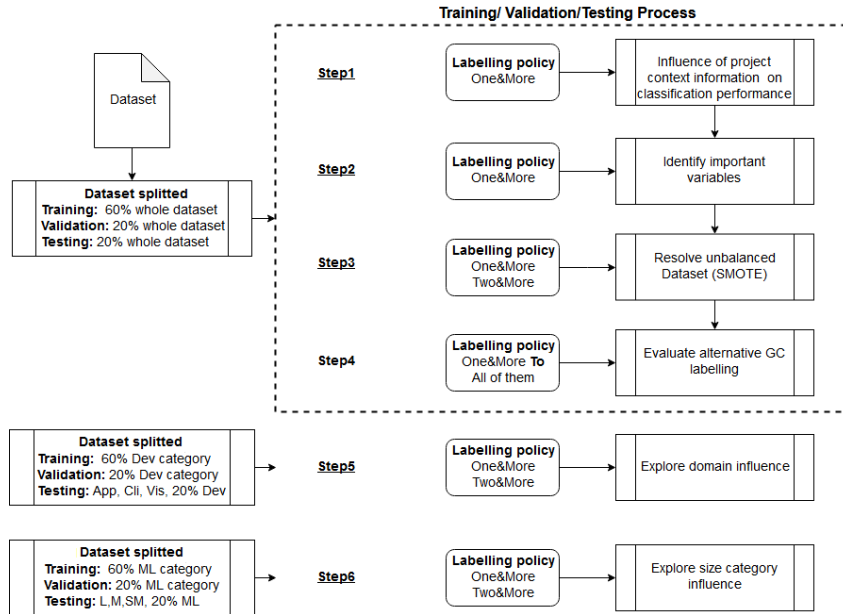


Fig. 2 Steps of the experimental work.

Step 1. In this step we look at the following question RQ.S1: Is the performance of classifiers affected by the use of project context information? We simplify this preliminary study considering only the criterion of one tool and more for *god class* labeling because the problem of unbalanced information is less severe. In order to answer RQ.S1, we train, validate and test the eight classifiers using all numeric variables (X_1, \dots, X_{16} , see Table 3) with and without the project context information (domain X_{17} and size category X_{18}). There are two null hypotheses for step 1: H_0^{Kappa} (resp. H_0^{MCC}) is that Kappa (resp. MCC) achieved by the classifiers without using project context information is greater or equal than using it, since our alternative hypothesis is that the performance of the classifiers improves using project context information. The comparison is performed according to the experimental methodology explained above. For the following steps, we select the configuration (with or without project context information) which provides the best classification performance (Kappa and MCC) over all the classifiers, according to a Wilcoxon test comparing the performances with and without project context information.

Step 2. Here, the question to explore is RQ.S2: Do we use unnecessary information which complicates the analysis without influence on the results? As in step 1, we use again one and more for *god class* labeling. First, we analyze the correlated variables, such as NOT (X_6) and NOTc (X_8), developing a test whose null hypothesis H_0^{NOT} is that classifiers achieve similar performance with and without the NOT variable. Two additional experiments train linear regressors with and without project context information in order to find co-linear variables. Finally, the importance of the variables X_1, \dots, X_{16} and of the project context information is estimated, and a set of important variables is selected and compared to the whole set of variables. Three tests are developed: 1) the null hypothesis $H_0^{ALL,IMP+PCI}$ is that classifiers achieve equal Kappa using all the variables (ALL) and using the important variables (IMP) plus project context information (PCI); 2) the null hypothesis $H_0^{ALL,IMP}$ is that classifiers achieve equal Kappa using all variables and using only the important variables (IMP); and 3) the $H_0^{IMP,IMP+PCI}$ is that classifiers achieve the same Kappa using important variables plus project context information and using just important variables. The best configuration (all variables, important variables or important variables plus project context information) is selected for the next step.

Step 3. Since the *god classes* represent a small fraction of the available classes, their detection can be considered an unbalanced classification problem. The synthetic minority over-sampling technique (SMOTE) (Chawla et al., 2002) is a popular oversampling method for unbalanced classification problems. This method balances both labels by creating artificial training patterns of the minority label (in our case, *god class*), in the set used to train the classifiers. The step 3 investigates RQ.S3: Is the SMOTE technique useful to reduce the unbalance problem? A first experiment evaluates the null hypothesis H_0^{SMOTE} . It states that classifiers, using one and more tools as labeling criterion, achieve greater or equal Kappa without SMOTE than with it.

Step 4. This step investigates RQ.S4: What effect does *god class* labeling have on our problem? This question is analyzed from two angles. First, RQ.S4.1: Is the classifier performance influenced by the labeling policy? Second, RQ.S4.2: Does the project context information increase the classifier performance in all the labeling policies? When classes are labeled as *god class* by one and more *design smell*

detection tools, it is expectable to have a high (resp. low) rate of false positives (resp. negatives). The increase in the number of tools required to label a class as *god class* change these rates and leads to a reduction in the number of *god classes*, so the classification problem becomes even more unbalanced. In order to take this fact into account, we develop experiments with different policies to combine the outputs of the detection tools. Specifically, we considered that a *god class* must be detected by two and more tools, by three and more tools, by four and more tools, and by the five tools (i.e., a logical AND of the five tools). To answer RQ.S4.1, we evaluate how these labelings influence the classifier performance. In order to answer RQ.S4.2, we compare the classifier performance without and with project context information using each of the previous alternative *god class* labelings. The null hypotheses of the tests associated to each experiment are listed in Table 6. Hypotheses H_0^i , with $i = 2, \dots, 5$, and H_0^{iC} , with $i = 1, 3, 4, 5$, correspond to RQ.S4.1, while H_0^{iW} , with $i = 1, \dots, 5$, correspond to RQ.S4.2.

Step 5. The steps 5 and 6 look separately at each of the parameters of the project context information. First, we check RQ.S5: Can we discard the influence of project domain (X_{17}) in classifiers performance? In order to measure its impact on the *god class* detection, each classifier is trained using 60% of the classes of domain Dev, and validated for parameter tuning using 20% of its classes. We select Dev for training because it is the only domain whose classes belong to all the available size categories (SM, M, ML and L), so the project size is not expected to bias the results of this experiment. Afterwards, the classifier is tested: 1) using the remaining 20% of the classes from the same Dev domain; 2) using the classes in the App domain; 3) using the classes in the Vis domain; and 4) using the classes in the Cli domain. The objective of this experiment is to check whether the classifier performance is different on the classes of the same domain (Dev) used during training and for the remaining domains, in which case the classification is strongly influenced by the domain information. For example, assuming that domain is not relevant for *god class* detection, a classifier trained for domain Dev should behave similarly on the remaining 20% of Dev classes and on the classes of domains App, Cli and Vis. Otherwise, we must assume that the domain information is relevant. Because we do not use the entire dataset in this step, we selected one and more tools and two and more tools as *god class* labeling criteria, because they provide more cases and because, as we will see in section 4.1, step 4 shows that the project context information improves the results with any criteria. The null hypotheses using one and more tools are that classifiers, trained using classes of domain Dev, achieve greater or equal performance in classifying classes from the App, Cli and Vis domains than with their own training domain, Dev. These hypotheses are denoted H_0^{DA1} , H_0^{DC1} , H_0^{DV1} respectively. Using two and more tools, the corresponding null hypotheses are denoted as H_0^{DA2} , H_0^{DC2} and H_0^{DV2} .

Step 6. Analogously to the previous step, we investigate RQ.S6: Can we discard the influence of project size (X_{18}) in classifiers performance? In order to answer this question, the classifiers are trained (resp. validated for parameter tuning) using 60% (resp. 20%) classes of size ML. Analogously to the previous step, we selected ML because it is the only size that contains classes of all the domains (App, Cli, Dev and Vis), in order to avoid that project domain may bias this experiment. The trained classifiers are tested: 1) using the remaining 20% of classes of size ML; 2) using classes of size SM; 3) using classes of size M; and 4) using classes of size L. Thus we evaluate whether the classifier performance is influenced by the class

Table 6 Null hypotheses of step 4 (PCI stands for project context information).

Name	Meaning
Classifiers achieve equal Kappa without PCI using One&More and using ...	
H_0^2	Two&More tools
H_0^3	Three&More tools
H_0^4	Four&More tools
H_0^5	Five tools
Classifiers achieve equal Kappa with PCI using Two&More and using ...	
H_0^{1C}	One&More tools (C means "context")
H_0^{3C}	Three&More tools
H_0^{4C}	Four&More tools
H_0^{5C}	Five tools
Classifiers achieve greater or equal Kappa without PCI than with it using ...	
H_0^{1W}	One&More tools (W means "with and without")
H_0^{2W}	Two&More tools
H_0^{3W}	Three&More tools
H_0^{4W}	Four&More tools
H_0^{5W}	Five tools

size. The same labeling criterion (one and more tools, and two and more tools) as in the previous step was taken for the same reasons. The null hypotheses with one and more tools are that classifiers, trained using size category ML, achieve greater or equal Kappa testing with projects classes in size category L ($H_0^{ML,L,1}$), size category M ($H_0^{ML,M,1}$), and size category SM ($H_0^{ML,SM,1}$), respectively, than with their own training size category, ML. With two and more tools, the analogous null hypotheses are denoted as $H_0^{ML,L,2}$, $H_0^{ML,M,2}$ and $H_0^{ML,SM,2}$.

4 Result and Discussion

The results of the steps 1-6 described in the previous section are presented here. Section 4.1 compares the classification results with and without project context information (step 1), with the whole variables and only important variables (step 2), with and without SMOTE (step 3) and with several policies of tool combination as oracle (step 4). Sections 4.2 and 4.3 discuss the experiments in steps 5 and 6 of the methodology performed for each factor (project domain and size category) independently.

4.1 Training Classifiers With the Whole Dataset

The left panel of Figure 3 plots the Kappa achieved by the classifiers with and without factors (blue and red lines, respectively), sorted by decreasing values, according to step 1 in the methodology. Excepting NB, with project context information the classifiers achieve higher Kappa (the red line is about 3% above the blue line). Therefore, the project context information, domain and size category (X_{17} and X_{18}) have a positive influence on *god class* detection. The best Kappa (above 60%) is achieved by RF using project context information. An asymptotic Wilcoxon-Pratt signed-rank test, and the same test with continuity correction, compare the Kappa values with and without the project context factors. The null

790 hypotheses H_0^{Kappa} can be rejected, with p -value 0.007812 ($r = 0.8416$), assum-
 791 ing the alternative hypothesis “Kappa values are greater using project context
 792 information”, so the difference is statistically significant with a large effect size
 793 ($r > 0.5$), which evidences that the project context information influences *god class*
 794 detection. The null hypotheses with MCC, H_0^{MCC} , is also rejected, with p -value
 795 0.01043. This is an example that Kappa and MCC lead to the same conclusion.
 796 For brevity, henceforth only p -values of tests applied on Kappa will be reported.

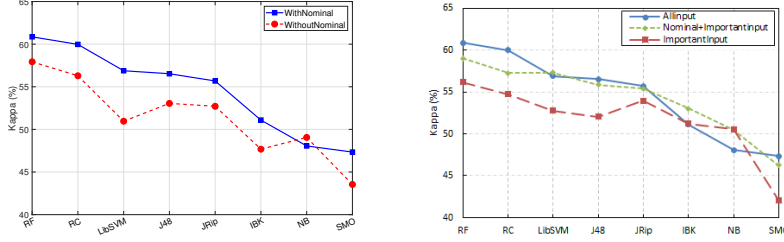


Fig. 3 Left: Kappa values of classifiers with and without project context information, sorted by decreasing Kappa values of the former to increase visibility. Right: Kappa values with all the variables, important variables and project context information, and important variables, sorted by decreasing values of the former.

Table 7 Kappa values (in %) achieved by the classifiers with all the variables (first row) and removing the NOT variable (second row).

	RF	RC	J48	JRip	IBK	NB	SMO	LibSVM
All variables	63.08	57.37	57.91	56.77	50.18	48.88	54.49	56.88
Without NOT	63.04	62.25	58.17	55.85	50.66	48.75	54.70	56.44

Table 8 Null hypotheses, p -values and effect size r of the tests in step 2.

Null hypothesis	p -value (effect size r)	
	Wilcoxon Test	Wilcoxon-Pratt
$H_0^{ALL,IMP+PCI}$	0.7422 (0.1485)	0.6744 (0.1485)
$H_0^{ALL,IMP}$	0.05469 (0.6931)	0.04995 (0.6931)
$H_0^{IMP+PCI,IMP}$	0.01562 (0.8416)	0.01729 (0.8416)

797 **Step 2** of methodology identifies the group of important input variables, dis-
 798 carding multi-collinearity problems that lead to misleading in the classification
 799 results using the whole input variable set. For example, some variables might be
 800 related to each other from the first view, such as TLOC, CLOC, NCLOC or project
 801 size (see Table 3 above for the meaning of variables). However, in *RefactorIT* the
 802 sum of CLOC plus NCLOC is not equal to TLOC due to blank lines. Besides,
 803 the project size is given by the criterion defined in (Fontana et al., 2012a) and

Table 9 Confusion matrix, Kappa and MCC (in %) of the best classifier (RF) using all input variables including project context information (PCI), left part, using SMOTE (S) with one and more tools, right part.

	RF (PCI)		RF (S, ≥ 1 tool)	
	Non-GC	GC	Non-GC	GC
Non-GC	80.17	2.79	78.49	4.47
GC	7.16	9.89	5.68	11.37
	Kappa(%)	MCC(%)	Kappa (%)	MCC (%)
	60.86	61.76	63.08	63.14

not by *RefactorIT*. In fact, the Pearson correlation between TLOC and NCLOC is 0.90. However, other variables are more correlated, such as NOT and NOTc (0.99), X_6 and X_8 resp., because most types are concrete (see Table 3). Table 7 reports the Kappa achieved by the classifiers with and without the NOT variable. The values are very similar for all the classifiers (a Wilcoxon test gives a p -value of 0.7422, and 0.6744 when asymptotic with continuity correction is applied). Thus, the null hypothesis H_0^{NOT} can not be rejected and the difference is not statistically significant, which means that indeed the variable NOT can be excluded, but the dependence between NOT and NOTc is not a problem for classification. The correlations between NOC (X_{13}) and NOT, and between NOC and NOTc, are lower (0.94). The remaining correlations between variables are below 0.9. On the other hand, we trained two linear regressors, with and without project context information, and we checked that the coefficients of both linear models are finite, so the collinearity problems are not very important.

Additionally, we determined the importance of each variable using the `rminer` package (Cortez, 2015) written in the R statistical computing language. Specifically, we selected as important variables those which importance, given by the `Importance` function of the `rminer` package, overcomes 0.5. Ten variables were selected: WMC, EXEC, RFC, TLOC, NOTc, DIT, NOA, NCLOC, DC and CLOC. The variables NOT and NOC were excluded due to their high correlations with NOTc, which was included. Surprisingly, the nominal variables size category and domain achieved low importance values and were also excluded. The reason is that `Importance` ranks variables according to its relevance, but variables with low relevance (domain and size among others, in this case) can still influence on the god class detection, as step 1 evidenced. The right panel of Figure 3 shows the classifier performance using all the variables, the important ones plus the project context information and only the important variables. A slightly better performance is achieved using the whole input set (blue line), followed very closely by the important variables with project context information (green line), and the worst Kappa values are achieved by the case of using just important variables without project context information (red line). Although the addition of project context information (green line) works similarly to the whole input set, the reduction in performance caused by feature selection, alongside with the low number (16) of features, suggests to keep all the variables for the next experiments. The differences between Kappa values of blue and green, blue and red, and green and red lines were analyzed with both a Wilcoxon and a Wilcoxon-Pratt test with continuity correction, used to avoid data tie warnings. Table 8 shows the null hypothesis, p -value and effect size (r) of each test. The difference between all the features (ALL) and the important features (IMP) with project context information

(PCI), denoted as IMP+PCI, is not significant (first row). This suggests that the use of PCI avoids a high performance reduction compared to all features, although the effect size is small ($r < 0.3$ in line 1 of Table 8). The difference between ALL and IMP (second row) is significant, so removing PCI the performance (Kappa) degrades significantly (IMP). Finally, the difference between PCI+IMP and IMP is also significant, so the PCI raises significantly the performance compared to IMP, in the last two cases with large effect size. These three tests evidenced that domain information brings the difference between the best performance (achieved with ALL or PCI+IMP, which are not significantly different) and alternatives without PCI such as IMP, which is significantly different with lower performance.

The left part of Table 9 reports the average confusion matrix over the 5 trials, the Kappa and MCC (both in %) achieved by the best classifier (RF) using all the variables. The diagonal values are high, but the percentage of non-detected *god classes* (false negatives) is relatively high (7.16%) compared to the percentage of *god classes* correctly detected (9.89%) although the false positive percentage is low (2.79%). The Kappa and MCC are high (60.86% and 61.76%, respectively). Since the performance degrades using the selected important features, in the following steps the experiments will use the whole set of metrics.

The experiment of **step 3** develops classification using SMOTE to balance both classes in the training set. Figure 4 shows that most classifiers obtained slightly better performances when the training set is balanced, excepting RC, LibSVM and IBK. The right part of Table 9 reports the confusion matrix of RF, which achieves the best Kappa, using SMOTE with one and more tools. Compared to the left part, the percentage of true positives is raised from 9.89% to 11.37%, while the false negatives reduces from 7.16% to 5.68%. However, the false positive arise from 2.79% to 4.47%, while Kappa and MCC raise slightly. The conclusion of the experiment is that classifiers only perform slightly better using SMOTE, because the difference between classifications with and without SMOTE is not statistically significant. A Wilcoxon test comparing both collections of Kappa values gives a p -value (resp. r value) of 0.1719 (resp. 0.3722), above 0.05 and a moderate effect size, between 0.3 and 0.5, so the hypothesis H_0^{SMOTE} of that performance without SMOTE is greater or equal to with it can not be rejected. However, given that class unbalancing will increase in the following experiments, the SMOTE method will be applied in order to achieve the best, although realistic, available performance.

According to **step 4**, we also developed experiments considering alternative class labels where a class is considered *god class* only when it is labeled as *god class* by: 1) three and more tools; 2) four and more tools; and 3) five tools. The objective of these experiments is to analyze the classifiers behavior related to the number of tools required to label a class as *god class*. Obviously, the more tools required to label a *god class*, the lower number of *god classes*, so the classification problem will be more unbalanced and the performance is expected to be reduced. From Table 5, the *god classes* vary from 15.55% of the classes with one and more tools to 0.19% with five tools. In other words, requiring less tools to detect a *god class* (e.g., by performing a logical OR of the tool outputs) the undetected *god class* rate is reduced, while requiring more tools (e.g., by performing an logical AND of the tool outputs) the false *god class* rate is reduced. In order to check this false positive reduction, a manual analysis of the *god class* detected was performed with the criteria of 4 tools and more and in the case of 5 tools. In the first analysis we detect 8 false positives among the 79 *god classes* and, in the second one, there

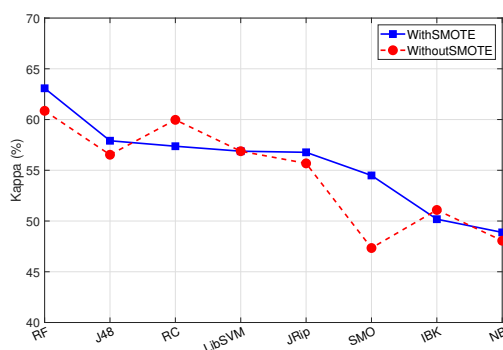


Fig. 4 Kappa values of classifiers with project context information using one and more tools with and without SMOTE, sorted by decreasing blue values.

is only one case, particularly strange, among 24 *god classes*, as mentioned in the introduction to the study design.

Figure 5 reports the Kappa of the different classifiers using the whole variables without (left panel) and with (right panel) project context information, SMOTE and one tool and more, two tools and more, three tools and more, four tools and more and five tools. The classifier's sortings by decreasing Kappa are very similar in both panels, being RC and RF the bests, followed by J48 and JRip. The Kappa values with one and more tools (black line) is the highest in the left plot, while the Kappa with two and more tools (blue line) is the highest in the right plot.

Inside each panel, a comparison among lines shows that Kappa degrades when the number of tools is increased. This degradation is faster without project context information (left plot) than with it (right plot). In fact, in the left plot the black, blue, red and gray lines are fairly separated, while in the right plot the blue, black and red lines are very near, the gray line is near them for the best classifiers, and the performance degrades dramatically with five tools (green line). This graceful degradation of Kappa on the right plot with increasing number of tools supports the positive contribution of project context information to the *god class* detection. The Wilcoxon test comparing the difference between the best series and the others in each panel produce similar results. By comparing in the left panel the best line (black) with the other lines, which gives p -values below 0.05 (statistically significant difference) and large effect size ($r > 0.5$) for red, gray and green lines are 0.007812, with $r = 0.8911$ for H_0^3 , H_0^4 and H_0^5 , which are rejected (the values are equal because all the points in each series are below the black line). In the right panel, the differences between the blue line (the best one) and the others are statistically significant for red, gray and green lines, with the same p and r values as in the left panel for H_0^{3C} , H_0^{4C} and H_0^{5C} , respectively, which are rejected, so these lines are clearly different. In both plots, blue, black and red lines achieve the best Kappa values. This suggests that using four and more and five tools (gray and green lines) reduces too much the number of *god classes*, leading to severely unbalanced problems where the conclusions of our study would be biased by the low classification performance.

Comparing both figures, which share the vertical scale, Kappa is higher on the right plot (i.e., with project context information) for each line, and the differences are statistically significant using One&More, Two&More, Three&More and Four&More tools and test has large effect size, with p -values (r) of 0.01172 (0.7921), 0.003906 (0.8911), 0.007812 (0.8416), 0.007812 (0.8416), respectively, for the null hypotheses H_0^{1W} , H_0^{2W} , H_0^{3W} and H_0^{4W} . The difference is not statistically significant with five tools, probably due to the strong unbalance problem, with p -value (r) of 0.05323 (0.5707), using a Wilcoxon-Pratt test due to ties, for H_0^{5W} , which can not be rejected. Nevertheless, the p -value is too close to 0.05.

Therefore, the following experiments are developed using one and more tools and two and more tools labeling policies.

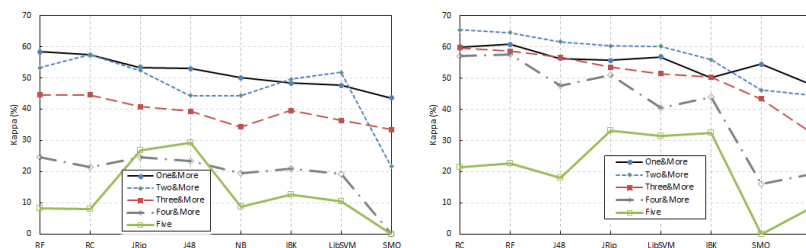


Fig. 5 Left panel: Kappa values of classifiers without project context information using SMOTE with one and more tools, two and more tools and so on. Right panel: Kappa values of classifiers with project context information using SMOTE with one and more tools, two and more tools and so on. Classifiers are sorted by decreasing Kappa of the best line.

4.2 Influence of Project Domain

We conducted the experiment described in **step 5** of the methodology using SMOTE and applying both criteria (one and more tools, two and more tools) separately on the domain category in order to test again the influence of domain information on the *god class* detection. Figure 6 shows the Kappa achieved by the classifiers when they are: 1) trained and tuned using Dev category (using 60% and 20% of Dev classes, respectively); 2) tested using the remaining 20% of Dev classes (black line), and using also the classes of App, Cli and Vis categories using one and more tools (left panel) and two and more tools (right panel). According to the left panel, the highest Kappa values are obtained in almost all the classifiers on the Dev category (SMO, LibSVM and NB are exceptions). Using the Wilcoxon test, the performance of the classifiers on the Dev domain in which they are trained is significantly superior to that obtained when used on the other App, Cli and Vis domains, presenting in all cases low p -values and large effect sizes (r): 0.01172 (0.792118), 0.01953 (0.7426) and 0.01953 (0.7426). Thus, their respective null hypotheses H_0^{DA} , H_0^{DC} and H_0^{DV} are rejected.

Table 10 (two left parts) reports the confusion matrix achieved by RF training and testing with Dev classes with one and more tools, and with two and more tools: the Kappa value using one and more tools (62.43%) is lower than the value

in the right part of Table 9 (63.08%), so despite of training and testing with classes of the same domain (Dev), the results are worse than using the whole dataset.

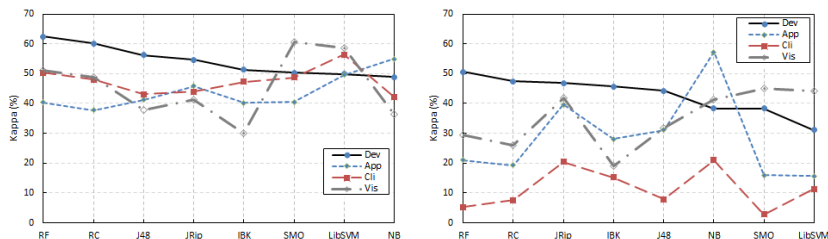


Fig. 6 Left: Kappa values of classifiers with domain information using one and more tools with SMOTE. Right: same plot using two and more tools. Classifiers are sorted by decreasing Kappa of the domain Dev.

Applying the second criterion (*god class* is detected by two and more tools, right part of Figure 6), the Kappa values are clearly lower than using one and more tools, but for domain Dev (black line) they overcome again the other domains for five of eight classifiers. This seems to confirm the influence of the project domain category on the *god class* detection. However, using two and more tools the Kappa values achieved by NB, SMO and LibSVM for domains App and Vis (blue and gray lines in the right panel of Figure 6) overcome the corresponding Dev values. We will see in the next subsection that this effect also happens, even with more intensity, with the project size category and NB, although this classifier, alongside with SMO and LibSVM, achieve the globally worst Kappa values (see subsection 5.3). The specific Kappa and MCC values are reported by rows 3-10 of Table 11 using SMOTE and two and more tools and varying the domain category (first two rows report the Kappa and MCC values using all the classes with project context information). Training with Dev domain, RF achieves the best result (Kappa 50.66%) on Dev testing, but NB outperforms this value for App (57.25%). Besides, the best Kappa for Cli and Vis are achieved by NB and SMO (21.05% and 44.99%, respectively), with values lower than Dev and App. The Wilcoxon test evaluates as significantly greater, with large size effects, the performance of classifiers in the ML domain than in the App and Cli domains with p -values and effect size (r) of 0.03906 (0.6436) and 0.003906 (0.003906) respectively, so their null hypotheses H_0^{DA2} and H_0^{DC2} are rejected, while H_0^{DV2} (Vis domain) is not rejected ($p = 0.125$, $r = 0.4456$). In this domain, the performance overcomes the Dev domain for classifiers NB, SMO and LibSVM because they perform poorly on the training domain (Kappa below 40%). On the other hand, we find particularly strange the performance of NB in domains App and Vis because it is often suboptimal in the remaining experiments, so its reliability in this case should be low.

For each of the remaining categories (App, Cli and Vis) we repeated the previous experiment but training the classifiers with domain categories other than Dev. For category App, we trained and tuned the classifiers using 60% and 20% of classes, respectively, and tested using the remaining 20% of the App classes. Afterwards, these classifiers are tested using the Cli, Dev and Vis classes. The process was repeated training with Cli and Vis instead of App, using one and more tools.

Table 10 Confusion matrix, Kappa, and MCC (in %) of the best classifier (RF, JRip) using all the input variables and SMOTE (S): training and testing with Software Development (Dev) domain with one tool and more (1T) and two tools and more (2T); training and testing with Medium Large (ML) size with 1T and 2T.

		RF(Dev, S, 1T)		RF(Dev, S, 2T)		RF(ML, S, 1T)		JRip(ML, S, 2T)	
	Non-GC	GC	Non-GC	GC	Non-GC	GC	Non-GC	GC	
Non-GC	83.18	3.37	95.82	1.03	84.80	3.27	95.51	1.92	
GC	4.94	8.5	1.68	1.47	4.83	7.09	1.5	1.08	
	Kappa (%)	MCC(%)	Kappa (%)	MCC (%)	Kappa (%)	MCC (%)	Kappa (%)	MCC (%)	
	62.43	62.59	50.66	62.53	59.3	59.10	36.91	37.07	

Table 11 Kappa (K) and MCC (in %) achieved by each classifier using SMOTE and two and more tools: with all project context nominal variables (upper part); training with Dev domain and testing with the domain categories (middle part); training with ML size and testing with the size categories (lower part). Best values are in bold.

Classifier	RF	RC	J48	JRip	IBK	NB	SMO	LibSVM
AllNominal-K(%)	64.67	65.55	56.1	61.72	60.44	56.03	46.25	60.21
AllNominal-MCC(%)	64.94	65.96	61.76	60.56	59.59	51.46	47.64	60.25
Dev-K (%)	50.65	47.36	44.3	46.91	45.71	38.28	38.26	31.08
Dev-MCC (%)	51.02	47.76	44.34	47.73	45.75	47.14	43.57	32.17
App-K (%)	20.95	19.29	31.09	39.61	28.07	57.25	15.96	15.68
App-MCC (%)	28.52	26.72	34.32	41.76	31.25	60.45	27.76	22.91
Cli-K (%)	5.33	7.6	7.79	20.46	15.22	21.05	2.82	11.39
Cli-MCC (%)	5.56	7.7	7.9	21.32	16.47	32.94	4.44	11.43
Vis-K (%)	29.49	25.97	31.66	41.73	18.98	41.21	44.99	44.06
Vis-MCC (%)	30.84	27.38	32.49	41.92	19.36	50.1	51.61	45.85
ML-K (%)	35.56	36.5	33.51	36.91	30.49	27.98	18.04	22
ML-MCC (%)	36.91	37.73	33.58	37.07	30.47	36.98	21.03	26.68
L-K (%)	23.52	23.02	30.11	32.58	30.3	57.48	28.29	11.67
L-MCC (%)	31.61	30.43	33.92	35.14	33.78	60.63	36.78	19.63
M-K (%)	15.53	15.32	22.97	28.29	24.41	59.39	8.22	20.67
M-MCC (%)	26.26	25	26.41	33.22	27.35	63.42	18.78	30.71
SM-K (%)	28.62	28.62	29.87	40.48	41.46	68.89	13.97	21.55
SM-MCC(%)	40.83	40.83	36.25	46.49	45.02	69.47	27.36	34.76

The results (not included here for brevity) showed that the Kappa values achieved by the classifiers trained with Dev category overcome the Kappa values achieved training with App, Cli or Vis categories. A possible explanation may be that these categories do not include information about some project size categories, so they depend on size as well as domain.

4.3 Influence of Project Size

In the experiment of **step 6** we selected the ML size category for the classifier training, while the test was conducted using the ML, SM, M and L categories. We follow the same strategy of step 5 replacing the Dev domain category by the ML size category, because it includes classes from App, Cli, Dev and Vis, so it should not be so influenced as the other size categories by the project domain. We also applied both criteria of detecting the *god class* (one and more tools, and two and more tools). The left panel of Figure 7 shows the Kappa values of each classifier testing with ML category (black line) and the remaining size categories using one and more detection tools. The black line overcomes the remaining lines,

1002 which evidences that the size factor should also be taken into account in *god class*
 1003 detection. The only exception is classifier NB, which overcomes ML for L and SM
 1004 size categories. Using one and more tools, the differences between ML and the other
 1005 size categories are statistically significant, and the Wilcoxon test gives *p*-values (*r*
 1006 values) of 0.01562 (0.8416), 0.007812 (0.8911), 0.01562 (0.8416) for L, M and SM
 1007 lines, respectively, so the corresponding null hypotheses $H_0^{ML,L,1}$, $H_0^{ML,M,1}$ and
 1008 $H_0^{ML,SM,1}$ are rejected. The third confusion matrix in Table 10 is achieved by RF,
 1009 which is the best classifier training and testing with ML size category and using one
 1010 and more tools. The value of Kappa (59.3%) is lower than the one achieved using
 1011 Dev (first confusion matrix in the same table), which empirically shows that the
 1012 ML category is not so significant for sizes as Dev category for domains. The rates
 1013 of false positives and negatives (3.27% and 4.83%, respectively) are only slightly
 1014 lower than their corresponding Dev values, so the worse behavior is explained by
 1015 the low number of *god classes* for ML categories compared to Dev domain, and by
 1016 the low true positive rate (7.09%) compared to Dev (8.5%) in Table 10.

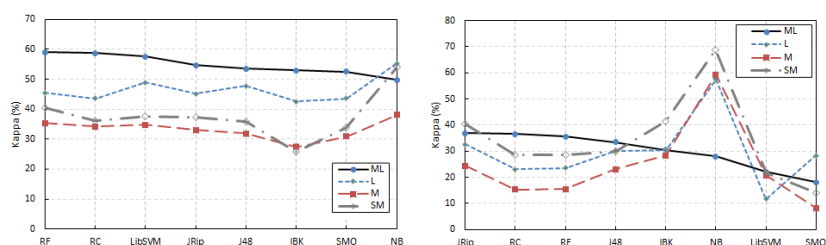


Fig. 7 Left panel: Kappa values achieved by the classifiers testing with ML and the other size categories using one and more tools with SMOTE. Right panel: Kappa values of classifiers with size information using two and more tools using SMOTE. Classifiers are sorted by decreasing Kappa of the ML size category.

1017 However, in the right panel (with two and more tools) of Figure 7 the black
 1018 line only overcomes the other lines for three classifiers, and for the remaining five
 1019 classifiers (JRip, IBK, NB, LibSVM and SMO) their lines are near or above the
 1020 black line, so the classifiers achieve equal or better Kappa testing on size categories
 1021 different to the one used for training. The Kappa values of NB in sizes L, M and
 1022 SM are specially high compared to ML, refusing our thesis that performance must
 1023 be higher in the training size category. However, the NB is one of the classifiers
 1024 which achieves globally the lowest Kappa values (see subsection 5.3). Therefore,
 1025 using two and more tools the results do not seem influenced by the project size
 1026 category. The confusion matrix of JRip, which is the best classifier training and
 1027 testing with ML size categories and two and more tools, is showed by Table 10
 1028 (right part): the Kappa value (36.91%) is much lower than using Dev domain
 1029 category (50.66%, second matrix in the same table), and the difference between
 1030 ML and Dev is much higher using two and more tools (50.66 – 36.91 = 13.75) than
 1031 using one and more tools (62.43 – 59.3 = 3.13). This means that the increase on
 1032 the classification unbalance caused by requiring two and more tools (from Table 5
 1033 *god class* percentages are 15.5% and 4.3% using ≥ 1 and ≥ 2 tools, respectively)

1034 affects more the significance of the ML size category than of the Dev domain
1035 category.

1036 The specific values of Kappa and MCC training with ML size classes and testing
1037 with ML and the remaining sizes are reported by rows 11-18 of Table [11](#). Again,
1038 these values support the comments of the previous paragraph, because using two
1039 and more tools the influence of size category it is not so clear as in the domain
1040 category case, and we cannot establish that ML is significantly greater than L,
1041 M and SM. The p -values and effect size (r) are 0.2305 (0.297), 0.09766 (0.4951)
1042 and 0.4727 (0.0495), respectively, on a Wilcoxon test, so the corresponding null
1043 hypotheses $H_0^{ML,L,2}$, $H_0^{ML,L,2}$ and $H_0^{ML,SM,2}$ can not be rejected. The reason
1044 of this lower influence of the size on the results, compared to domain, may be
1045 the ordinal nature of the size category. Besides, the criterion to label a class with
1046 a given size category is not so crisp as with domain categories. Analyzing the
1047 classifier Kappa and MCC for each size category and two and more tools (lower
1048 block of Table [11](#)), the RC achieves the best Kappa (37.73%) testing with ML,
1049 but NB is the best for the other size categories with higher Kappa values (57.48%,
1050 59.39% and 68.89% for L, M and SM, respectively), so the *god class* detection
1051 works better testing on size categories other than the training size category.

1052 Analogously to the domain case, we trained classifiers with the L, M and SM
1053 categories separately, testing them with the same and different size categories
1054 and one and more tools. The classifiers trained on these categories exhibited lower
1055 performance than those trained with ML size category (the results are not included
1056 here for brevity). This arises that results are better when only one factor (in this
1057 case, the project size) is taken into account.

1058 5 Replication

1059 In this section a replication of Step 1 of the previous study is presented. The
1060 general *RQ* applies, and particularly the specific Step 1 question *RQ.S1* is adapted
1061 as *RQ.S1.rep*.

1062 *RQ.S1.rep*: Is the performance of classifiers affected by the use of project
1063 context information? using a new dataset in which a human labeling policy has
1064 been followed?

1065 We use exactly the same null hypotheses H_0^{Kappa} for Step 1 of the previous
1066 study: is Kappa achieved by the classifiers without using project context informa-
1067 tion greater or equal than using it?

1068 The Machine Learning techniques are the same used in the previous study.
1069 The rest of the steps in the methodology do not apply because of the nature of
1070 the dataset labeling. The new data collection is described in the next subsection.

1071 5.1 Data collection

1072 The new dataset is generated on the basis of a dataset available in the current
1073 literature and previously used in [\(Pecorelli et al., 2019, 2020\)](#).

1074 This dataset consists of manually validated code smells instances of 125 releases
1075 of 13 projects. The original dataset [\(F. et al., 2019\)](#) is modified in order to 1)

select a release of each project; 2) select *god class* labeling; 3) include the project context information (domain and size category), and 4) include the metrics used in the study described in Section 3. The new dataset is available at the same url mentioned before: <https://citius.usc.es/investigacion/datasets/project-nominal-information>.

The release of each project was selected using the one with more *god classes*. In our case has no sense to work with different releases of the same project because we are interested in study the influence of the project context and this is the same from one release to other.

Table 12 shows a description of the dataset collected in this replication and the number of *god classes* labeled in each project.

Table 12 Characteristics of the projects in the replication dataset. (Dev: Software Development; Vis: Diagram Generator/Data Visualization; Cli: Client Server Software; XL: Very Large; L: Large; ML: Medium-Large; M: Medium.)

Project name	Domain	Size Category (TLOC)	NOC	(GCs)
ant-rel-1.8.3	L (119256)	Dev	1473	6
argouml-VERSION_0_14	L (199075)	Vis	1373	2
cassandra-cassandra-1.I.0	L (110712)	Cli	699	2
apache-wicket-1.4.11	L (174033)	Dev	1568	4
derby-10.3.3.0	XL (535187)	Cli	1746	24
hadoop-release-0.2.0	M (34662)	Dev	327	2
hsqldb-2.2.0	L (254014)	Cli	590	11
incubator-livy-0.6.0-incubating	L (130696)	Cli	1016	6
nutch-release-0.7	ML (50578)	Cli	532	0
qpid-0.18	L (189271)	Cli	2172	6
xerces-Xerces-J_1_4_2	L (150445)	Dev	489	6
eclipse-R3_4	L (423423)	Dev	5061	25
elasticsearch-v0.19.0	L (315619)	Cli	1395	2
Total			18441	96

The classification in domains was conducted in the same way the previous study. In this case, no agreement was reach in first round with incubator-livy and hadoop projects. In second round with open discussion was detected a problem with the url describing the incubator-livy project and an agreement was reached after the four authors consult the same project documentation. No agreement was reached in the case of hadoop in second round. Two hadoop experts external to this study. Both indicates the same classification and finally the agreement was reached taking into account the experts opinion. The fifth author, as in the previous study, review, check and agree.

5.2 Results and discussion of the replication

The classifiers used in the previous experiments (see Table 3 in section 3.2) are applied on the dataset described in Subsection 5.1, including the project context information.

Figure 8 (left panel) shows that the best Kappa values are obtained over the classifiers trained with the project context information, although several of them achieve values below 40%. In this dataset the number of *god classes* labeled by human experts are 96 of a total of 18,441 classes, which represents 0.52% of true positives. This percentage is slightly lower than the one using 4 tools and more and

higher than using 5 tools and more in out dataset (see Table 5). The degradation shown in the Kappa achieved by the classifiers with and without project context information in step 4 of our methodology led us to discard this labeling in our final analyses. We also applied the classifiers after a re-sampling with SMOTE, achieving similar results.

The Figure 8 (right panel) shows that the average performance decreases with the number of *god classes* marked by the labeling policy (one, two, three, four and five tools and more, and human). In all the policies, the blue box plot (with project context information) is above the red box plot (without project context information). Using the human labeling (right end of the plot), the blue (resp. the red) box plot is similar to (resp. much higher than) using 4 tools and more. The Kappa ranges achieved by the different classifiers (i.e., the box plot height) also change by reducing the number of *god class*. When we have enough examples all the algorithms offer consistently close results (narrow box plots), but when the true examples are reduced the Kappa obtained by the different algorithms are more arbitrary with larger ranges (wider box plots). In addition, the *god classes* are not distributed across all the categories used in previous experiments, so only the initial analysis has been carried out with all the information.

In order to review the question RQ.S1.rep (is the performance of classifiers affected by the use of project context information?) over the human labeled dataset, we conducted a new Wilcoxon test on the Kappa values of the figure. The null hypothesis is that the classifiers trained without project context information achieve Kappa greater than or equal to those trained with project context information. The p -value 0.2734 of this test does not allow us to reject the null hypothesis and assume the relevance of the project context information in improving the performance of the classifiers, although the effect size of the test is moderate ($r=0.2475$). This also happened using 5 tools and more, despite classifiers with better Kappa (excepting RC) achieve higher Kappa trained with project context information (see Figure 8, left panel). Looking at the right panel of Figure 8, the Kappa achieved using project context information is always greater than or equal to the one achieved without project context information. This is confirmed by a Wilcoxon test defined by the null hypothesis: the average performance of all the algorithms without using project context information is higher or equal than when using it? This test provides a p -value of 0.015 with a large effect size ($r=0.78$).

5.3 Joint analysis by classifier

Table 13 reports the Kappa median and mean, and Friedman rank (Demšar, 2006) of each classifier, calculated over all the experiments developed in the paper. The first row labels the classifier positions given by their Friedman rank. The RF achieves the best position, which corresponds to the lowest rank (2.6), which means that RF is between positions 2 and 3 in average over all the experiments. Note that RF achieves the best Kappa in most experiments (see e.g. Figures 3, 4, 5, excepting Four&More and Five tools, left panels of Figures 6 and 7), and the only exceptions are experiments with severe class unbalance (right panels in Figures 6 and 7). The RF also achieves the best Kappa median (51.0%) and mean (45.5%), tied with JRip (Table 13). A group of classifiers with ranks between 3.6 and 4.4 includes RC, JRip, J48 and LibSVM. Another group includes NB and IBK (with ranks

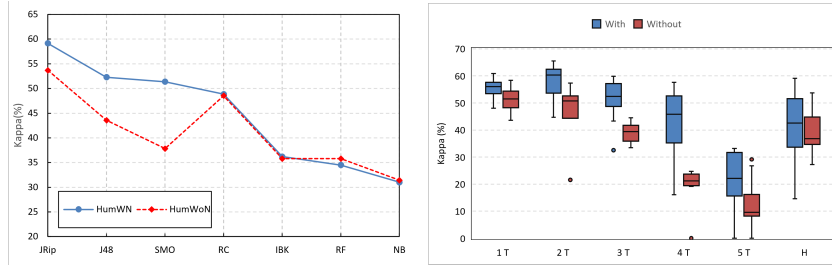


Fig. 8 Left panel: Kappa achieved by classifiers with and without project context information using the *god class* identification of human experts, sorted by decreasing blue values. Right panel: box plot of Kappa for values of classifiers with (blue) and without (red) project context information with labeling polices one and more tools (1T) and so on, and with human labeling (H).

5.4-5.5), while SMO achieves the worst rank, with the lowest performance. Please note that the classifiers with the best performance (and lower ranks) are those that support, in the previous experiments, our hypothesis that project context information is important for *god class* detection. Conversely, those classifiers with lower ranks (e.g. NB, IBK and SMO) are the ones which support in a lower degree these hypotheses and exhibit erratic behavior.

Table 13 Kappa median and mean, and Friedman rank achieved by each the classifier over all the experiments.

Position	1	2	3	4	5	6	7	8
Method	RF	RC	JRip	J48	LibSVM	NB	IBK	SMO
Median	51.0	48.7	46.9	44.3	49.6	48.1	45.7	43.3
Mean	45.5	44.3	45.5	43.3	42.4	42.8	40.6	34.8
Rank	2.6	3.6	3.6	4.1	4.4	5.4	5.5	6.8

1157 6 Threats to Validity

1158 This section discusses threats to the validity in the context of the conducted ex-
 1159 periments. Construct validity is concerned with the relationship between theory
 1160 and observation. Internal validity relates to any negative effects on the experi-
 1161 ment design, while external validity relates to the significance of the experiment
 1162 outcomes.

1163 6.1 Construct Validity

1164 The selection of the tools and techniques are probably the main threat to the con-
 1165 struct validity of the experiment. Firstly, we selected five software tools *DECOR*,
 1166 *PMD*, *JDeodorant*, *Together* and *iPlasma* which allow *god class* detection, work
 1167 with Java projects extracted from the SourceForge repository, are available for

1 1168 us and are widely used in the literature. These tools are completely independent
2 1169 because they are developed by different research groups in Software Engineering.
3 1170 Secondly, the metrics are evaluated using *RefactorIT*, a software tool which is in-
4 1171 dependent from the previous ones and is widely used in the literature. We did not
5 1172 use several tools for metric evaluation because, although their metric values and
6 1173 the classifier performance for *god class* detection might differ among tools, the con-
7 1174 clusions of the experiments (e.g., influence of the project context information for
8 1175 *god class* detection) should be similar. Thirdly, we selected a collection of classi-
9 1176 fiers representing the current state-of-the-art, so it is not expectable that classifiers
10 1177 excluded from this collection exhibit a behavior different to our experiments. Fi-
11 1178 nally, the distribution of projects among domains may seem biased at first glance,
12 1179 given the low number of projects in domains Cli and Vis. However, note that this
13 1180 unbalance reflects the natural distribution of projects across domains, because
14 1181 the selection of projects was random. Therefore, selecting an artificial balanced
15 1182 distribution would not be realistic, biasing the results of the experimentation for
16 1183 sure.

1184 6.2 Internal Validity

1185 The main threat to internal validity relates to the lack of consensus in the literature
1186 on the correct definition of the *god class design smell*. We managed this threat
1187 by using a committee of five different *god class* detection tools (open source and
1188 commercial), which are based on different detection strategies, to label each class
1189 as *god class* or not. Some of these tools, such as DECOR, include the definition
1190 of the anti-pattern *god class* which generates the detection code automatically
1191 from a definition of the anti-pattern expressed in a DSL and considering different
1192 heuristics.

1193 We used several policies to combine the single labels in order to increase the
1194 reliability of the committee label (e.g., to avoid false *god class* labels), and to
1195 evaluate how the conclusions of the experiments hold for each policy (step 4 in our
1196 methodology). Since the results (e.g., the influence of project context information)
1197 are consistent among policies (except requiring ≥ 4 tools, due to the low *god class*
1198 rate), it is expectable that these results would also be consistent with a manual
1199 *god class* labeling issued by an expert, but such an experimentation falls outside
1200 the scope of the current work.

1201 Another threat is related to the low number of *god classes*, which leads to
1202 an unbalanced classification problem. We minimized the impact of this unbalance
1203 on *god class* detection using a well-known oversampling technique called SMOTE
1204 (step 3 in our methodology). The last threat involves the potential relations of
1205 dependence and redundancy among class metrics, alongside with the irrelevance
1206 of some metrics for *god class* detection, a threat that is already analyzed in step
1207 2. With respect to project context information (domain and size category), whose
1208 relevance for *god class* detection is explored in the current work, the threat involves
1209 the subjectivity of the project categorization in terms of domain and size, and the
1210 bounds in the number of categories. To overcome this threat in relation to the
1211 domain categories, we used the same classification as (Fontana et al., 2013a),
1212 which was initially proposed by (Tempero et al., 2010) for the Qualitas Corpus.

1
2 1213 As regards the size categories, we considered in the current work the ones defined
3 1214 by (Fontana et al., 2013a).

4 5 1215 6.3 External Validity

6
7 1216 Some decisions taken during the experiment construction might constrain the gen-
8 1217 eralization of results. All the software tools used for *god class* detection worked
9 1218 with Java projects, specifically from the SourceForge repository, so we restricted
10 1219 to the open source SourceForge projects due to availability reasons. We randomly
11 1220 selected projects from the repository until there is at least one project of each
12 1221 domain and size category, achieving a collection which represents the distribution
13 1222 of projects in the repository. The experiments involved a high number of classes
14 1223 of each domain and size, although the number of projects is low for some domains
15 1224 and sizes, and this might be another threat to the validity of the conclusions. As
16 1225 a threat to generalization it is possible that other results arises when experiment-
17 1226 ing with projects of other domains, age, and not open source. Nevertheless, the
18 1227 exploratory study results are promising. Hence, a large-scale study that avoid this
19 1228 generalization threats should be developed. Finally, the experiments can be hardly
20 1229 extrapolated to other *design smells*, because the selected tools or the metrics used
21 1230 may not be adequate to detect the new *design smells*. Besides, other problems sim-
22 1231 ilar to class unbalance might appear for other *design smells*, which would require
23 1232 an specific analysis.

24 25 26 27 1233 7 Conclusion and Future Work

28
29 1234 In this work, we study the influence of two factors related to the project con-
30 1235 text information on the automatic detection of *god class design smell*. Specifi-
31 1236 cally, we aimed to investigate whether machine learning classifiers increase their
32 1237 performance for *god class* detection when some project context information, i.e.
33 1238 domain and size category, is included. To this end, we designed an exploratory but
34 1239 broad study to ascertain whether these two project factors are relevant to eight
35 1240 well-known classifiers, and whether they would affect the effectiveness agreement
36 1241 (Kappa) of the classification outcomes. In order to evaluate whether the conclu-
37 1242 sions depend on the classifier, we selected a collection of classifiers including the
38 1243 support vector machine with Gaussian kernel, the k-nearest neighbor classifier, the
39 1244 C4.5 decision tree, association rules with reduced error pruning, sequential min-
40 1245 imal optimization, naive Bayes classifier, random committee and random forest.
41 1246 For the experiments, we created a large dataset consisting of 12,587 classes from 24
42 1247 Java software projects with different size categories and domains. We selected five
43 1248 commonly used *design smell* detection tools to automatically detect the *god classes*
44 1249 in the dataset, and developed an analysis of the class metrics importance for fea-
45 1250 ture selection. Since the classification problem is unbalanced by nature, we used
46 1251 the synthetic minority over-sampling technique (SMOTE) in order to increase the
47 1252 classifier performance. We also studied the validity of our experiments considering
48 1253 several labeling policies to combine the five software tools for *god class* detection.
49 1254 According to the Kappa values, in our first experiment with the whole dataset and
50 1255 only one labeling policy (one tool and more), the null hypothesis H_0^{Kappa} of that

performance without project context information is greater or equal than with it is rejected. This evidences the influence of project context information in this case, giving a positive answer to the research question RQ.S1. When analyzing whether we use unnecessary information (RQ.S2), we discarded collinearity between variables that make then usefulness. We also analyzed the variable importance to see if some variable is not useful for *god class* detection, but surprisingly the project context information was identified as low important. However, when analyzing their influence through the null hypotheses $H_0^{ALL,IMP+PCI}$, $H_0^{ALL,IMP}$ and $H_0^{IMP,IMP+PCI}$, each comparing the performance using the groups of variables indicated in their names, we found that the selection of the most important features only degrades significantly the results when the project context information is excluded. Thus, all the available information must be considered useful, giving a negative answer to the RQ.S2.

Since the *god class* detection is an unbalanced classification problem, we used SMOTE to balance the training set. Although we can see in Figure 4 that the performance increases slightly, the null hypotheses H_0^{SMOTE} of that performance without SMOTE is greater or equal to with it can not be rejected. As a consequence, we cannot obtain empirical evidence of the usefulness of the technique in this case, giving a negative answer to RQ.S3.

In the step 4 of our methodology we reviewed the influence of *god class* labeling policies, specifically of the number of tools required to identify a *god class*, on the classification performance, since the class unbalancing increases when more tools are required. Through the null hypotheses listed in Table 6 we compare the classifier performances achieved by the best policy and by the others. We found that the same hypothesis are rejected with $(H_0^3, H_0^4 \text{ and } H_0^5)$ and without $(H_0^{3C}, H_0^{4C} \text{ and } H_0^{5C})$ project context information so the RQ.S4.1 (do the labeling policy influences the performance?) can be answered positively because the performance of the classifiers changes strongly from using 3 tools and more regardless the project context information is used or not.

Regarding RQ.S4.2 (does the project context information increase the classifier performance in all the labeling policies?), the answer is also positive, because the null hypotheses that performance without project context information is greater or equal than with project context information is rejected in all the labeling policies $(H_0^{iW}, \text{ with } i = 1, \dots, 4)$ excepting with five tools (H_0^{5W}) due to the huge class unbalancing. This analysis is complemented with the study carried out on the replication with the second dataset labeled by human experts. In this case the hypothesis about the influence of the project context information on the performance of the classifiers could not be confirmed. This study also shows that the average performance of classifiers with project context information is better than without project context information. Finally, the average performance of the algorithms and their dispersion is consistent with the degradation observed in Step 4 when the number of *god classes* is reduced.

Finally, when the project domain and size categories are studied separately, the performance of the classifiers is reduced for domain categories that were not present in the classifier training. This effect is weaker for size categories when a labeling policy requiring two and more *god class* detection tools is applied. In both cases, we observe changes in the classifiers performance, so we cannot rule out the influence of these parameters and questions RQ.S5 (can we discard the influence of project domain on the classifiers performance?) and RQ.S6 (the analogous for project

size) are answered negatively. Therefore, we conclude that project domain and size are important for detecting the *design smell* in the context of the classes that are analyzed through machine learning classifiers, and they should be taken into account in future works to obtain a more accurate *god class* detection. However, the project domain exhibits stronger influence compared to project size, whose influence is not so clear using two and more tools for *god class* identification. The weakness of the influence of nominal project size may be affected by also using project size as an ordinal feature. In fact, step 2 points out TLOC as one of the most important characteristics.

We have seen how project context information influences all the experiments: RQ.S1, RQ.S2 and RQ.S4 reflect that performance changes significantly when considering the project context information regardless of the labeling used, except for the most extreme situation (five tools and more labeling policy); besides, RQ.S5 and RQ.S6 also show that influence of domain and size can not be discarded independently for each factor. Consequently, the research question RQ (does the project context information influence the *god class* detection?) can be answered positively.

Taking these factors (project domain and size category) into account is part of adaptation to the context which should be investigated in a more general sense in future work, including other project context information. Project domain is well known before starting the project development but size category is particularly interesting to be researched in future work combined with methods for predicting project size category before starting the project. This can help developers from the very beginning to adequate *god class* detection to the project context.

Considering classifiers, the random forest and random committee achieve the best performance in most experiments, exhibiting stable and consistent behaviors. On the contrary, sequential minimal optimization and naive Bayes report the poorest performances with lack of agreement to the other classifiers.

Regarding other project context factors that can influence on detection there are some possibilities to study such as the software development process, as various authors suggests. Although in (Crespo et al., 2006) the influence of the type of software such as application, framework, library was analyzed and found no evidence of this project characteristic on relative metric's thresholds. Nevertheless, an specific and larger study can be conducted in order to check whether it influences in design smell detection, including SPLs as another type of software construction to analyze. It is also important to study the presence of automatically generated parts of software in some projects and how can it influences in design smell detection.

In addition to the former, our future work will focus on replicating the experiments reported in this work, using different datasets labeled by experts involving professional developers and QAs from the industry in identifying true *god classes*. We also believe that our findings can guide developers to better focus their efforts to identify other *design smells*.

References

- Alkharabsheh K, Crespo Y, Manso E, Taboada J (2016a) Comparación de herramientas de detección de design smells. In: Jornadas de Ingeniería del Software

- 1
2 1351 y Bases de Datos, pp 159–172
- 3 1352 Alkharabsheh K, Crespo Y, Manso E, Taboada J (2016b) Sobre el grado de acuerdo
4 1353 entre evaluadores en la detección de design smells. In: Jornadas de Ingeniería
5 1354 del Software y Bases de Datos, pp 143–157
- 6 1355 Alkharabsheh K, Crespo Y, Manso E, Taboada J (2018) Software Design Smell
7 1356 detection: a systematic mapping study. *Software Quality Journal* DOI 10.1007/
8 1357 s11219-018-9424-8, URL <http://dx.doi.org/10.1007/s11219-018-9424-8>
- 9 1358 Alves TL, Ypma C, Visser J (2010) Deriving metric thresholds from benchmark
10 1359 data. In: Proceedings of the 2010 IEEE International Conference on Software
11 1360 Maintenance, IEEE Computer Society, ICSM '10, pp 1–10, DOI 10.1109/ICSM
12 1361 .2010.5609747
- 13 1362 Azadi U, Fontana FA, Taibi D (2019) Architectural smells detected by tools: A
14 1363 catalogue proposal. In: Proceedings of the Second International Conference on
15 1364 Technical Debt, IEEE Press, TechDebt '19, p 88–97, DOI 10.1109/TechDebt.2
16 1365 019.00027
- 17 1366 Azeem MI, Palomba F, Shi L, Whang Q (2019) Machine learning techniques for
18 1367 code smell detection: A systematic literature review and meta-analysis. *Informa-*
19 1368 *tion and Software Technology* 108:115–138
- 20 1369 Bekkar M, Djemaa DK, Alitouche DA (2013) Evaluation measures for models
21 1370 assessment over imbalanced data sets. *Journal of Information Engineering and*
22 1371 *Applications* 3(10)
- 23 1372 Blackman N, Koval J (2000) Interval estimation for Cohen's kappa as a measure
24 1373 of agreement. *Statistics in Medicine* 19(5):723–741
- 25 1374 Borland (2008) Together. <http://www.borland.com/together>
- 26 1375 Brown WH, Malveau RC, McCormick HW, Mowbray TJ (1998) *AntiPatterns:*
27 1376 *refactoring software, architectures, and projects in crisis.* John Wiley & Sons,
28 1377 Inc.
- 29 1378 Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) SMOTE: synthetic
30 1379 minority over-sampling technique. *J Artif Intel Res* 16:321–357
- 31 1380 Choinzon M, Ueda Y (2006) Detecting defects in object oriented designs using
32 1381 design metrics. In: *J. Conf. on Knowledge-Based Software Engineering*, pp 61–
33 1382 72
- 34 1383 Cohen J (2013) *Statistical power analysis for the behavioral sciences.* Academic
35 1384 press
- 36 1385 Copeland T (2005) *PMD applied.* Centennial Books
- 37 1386 Cortez P (2015) A tutorial on using the rminer r package for data mining tasks.
38 1387 Tech. rep., Univ. do Minho. Escola de Engenharia
- 39 1388 Counsell S, Mendes E (2007) Size and frequency of class change from a refactoring
40 1389 perspective. In: *Int. Conf. on Software Evolvability*, pp 23–28
- 41 1390 Crespo Y, López C, Marticorena R (2006) Relative thresholds: Case study to in-
42 1391 corporate metrics in the detection of bad smells. In: *Proceedings of 10th ECOOP*
43 1392 *Workshop on Quantitative Approaches in Object-Oriented Software Engineer-*
44 1393 *ing, Universita della Svizzera italiana Press*, pp 109–118
- 45 1394 Demšar J (2006) Statistical comparisons of classifiers over multiple data sets. *J*
46 1395 *Mach Learn Res* 7:1–30, URL [http://dl.acm.org/citation.cfm?id=124854](http://dl.acm.org/citation.cfm?id=1248547.1248548)
47 1396 [7.1248548](http://dl.acm.org/citation.cfm?id=1248547.1248548)
- 48 1397 Di Nucci D, Palomba F, Tamburri DA, Serebrenik A, De Lucia A (2018) Detecting
49 1398 code smells using machine learning techniques: are we there yet? In: *Intl. Conf.*
50 1399 *on Software Analysis, Evolution and Reengineering*, pp 612–621
- 51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- 1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
- 1400 F P, D DN, C DR, A DL (2019) A large empirical assessment on the role of data
1401 balancing in machine-learning-based code smell detection - online appendix.
1402 URL <https://figshare.com/s/5da162e21b8d54fbfce8>
1403 Fernandes E, Oliveira J, Vale G, Paiva T, Figueiredo E (2016) A review-based
1404 comparative study of bad smell detection tools. In: Proc. 20th Intl. Conf. on
1405 Evaluation and Assessment in Software Engineering, p 18
1406 Fokaefs M, Tsantalis N, Chatzigeorgiou A, Sander J (2009) Decomposing object-
1407 oriented class modules using an agglomerative clustering technique. In: 2009
1408 IEEE International Conference on Software Maintenance, pp 93–101, DOI 10.1
1409 109/ICSM.2009.5306332
1410 Fontana FA, Mariani E, Mornioli A, Sormani R, Tonello A (2011) An experience
1411 report on using code smells detection tools. In: Intl. Conf. on Software Testing,
1412 Verification and Validation Workshops, pp 450–457
1413 Fontana FA, Braione P, Zanoni M (2012a) Automatic detection of bad smells in
1414 code: An experimental assessment. *J Object Technology* 11(2):5–1
1415 Fontana FA, Braione P, Zanoni M (2012b) Automatic detection of bad smells in
1416 code: An experimental assessment. *Journal of Object Technology* 11(2):5:1–38,
1417 DOI 10.5381/jot.2012.11.2.a5
1418 Fontana FA, Ferme V, Marino A, Walter B, Martenka P (2013a) Investigating
1419 the impact of code smells on system’s quality: An empirical study on systems
1420 of different application domains. In: Intl. Conf. on Software Maintenance, pp
1421 260–269
1422 Fontana FA, Zanoni M, Marino A, Mantyla MV (2013b) Code smell detection:
1423 Towards a machine learning-based approach. In: Int. Conf. on Software Main-
1424 tenance, pp 396–399
1425 Fontana FA, Ferme V, Zanoni M, Yamashita A (2015) Automatic metric thresholds
1426 derivation for code smell detection. In: IEEE/ACM 6th International Workshop
1427 on Emerging Trends in Software Metrics (WETSoM), IEEE Computer Society,
1428 Los Alamitos, CA, USA, pp 44–53, DOI 10.1109/WETSoM.2015.14
1429 Fontana FA, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experi-
1430 menting machine learning techniques for code smell detection. *Empirical Soft-
1431 ware Engineering* 21(3):1143–1191
1432 Fourati R, Bouassida N, Abdallah H (2011) A metric-based approach for anti-
1433 pattern detection in UML designs. *Computer and Information Science* pp 17–33
1434 Fowler M, Beck K (1999) Refactoring: improving the design of existing code.
1435 Addison-Wesley Professional
1436 Gronback RC (2003) Software remodeling: Improving design and implementation
1437 quality, using audits, metrics and refactoring in borland together controlcenter.
1438 Tech. rep., A Borland White Paper
1439 Guggulothu T, Moiz SA (2020) Code smell detection using multi-label classifica-
1440 tion approach. *Software Quality Journal* 28:1063–1086
1441 Hall T, Beecham S, Bowes D, Gray D, Counsell S (2011) Developing fault-
1442 prediction models: What the research can show industry. *IEEE Software*
1443 28(6):96–99
1444 Hassaine S, Khomh F, Guéhéneuc YG, Hamel S (2010) Ids: an immune-inspired
1445 approach for the detection of software design smells. In: Intl. Conf. Quality of
1446 Information and Communications Technology, pp 343–348
1447 Herbold S, Grabowski J, Waack S (2011) Calculation and optimization of thresh-
1448 olds for sets of software metrics. *Empirical Software Engineering* 16(6):812–841,

- 1
2 1449 DOI 10.1007/s10664-011-9162-z
3 1450 Hollander M, Wolfe DA, Chicken E (2013) Nonparametric statistical methods, vol
4 1451 751. John Wiley & Sons
5 1452 Khomh F, Vaucher S, Guéhéneuc YG, Sahraoui H (2011) BDTEX: A GQM-based
6 1453 Bayesian approach for the detection of antipatterns. *J Systems and Software*
7 1454 84(4):559–572
8 1455 Kreimer J (2005) Adaptive detection of design flaws. *Electronic Notes in Theoret-*
9 1456 *ical Computer Science* 141(4):117–136
10 1457 Lanza M, Marinescu R (2007) Object-oriented metrics in practice: using software
11 1458 metrics to characterize, evaluate, and improve the design of object-oriented sys-
12 1459 tems. Springer Science & Business Media
13 1460 Li W, Shatnawi R (2007a) An empirical study of the bad smells and class error
14 1461 probability in the post-release object-oriented system evolution. *J Systems and*
15 1462 *Software* 80(7):1120–1128
16 1463 Li W, Shatnawi R (2007b) An empirical study of the bad smells and class error
17 1464 probability in the post-release object-oriented system evolution. *Journal of Sys-*
18 1465 *tems and Software* 80(7):1120–1128, DOI [https://doi.org/10.1016/j.jss.2006.10.](https://doi.org/10.1016/j.jss.2006.10.018)
19 1466 018
20 1467 Linares-Vásquez M, Klock S, McMillan C, Sabané A, Poshyvanyk D, Guéhéneuc
21 1468 YG (2014) Domain matters: Bringing further evidence of the relationships
22 1469 among anti-patterns, application domains, and quality-related metrics in java
23 1470 mobile apps. In: *Proceedings of the 22Nd International Conference on Program*
24 1471 *Comprehension, ACM, ICPC 2014*, pp 232–243, DOI 10.1145/2597008.2597144
25 1472 Liu H, Liu Q, Niu Z, Liu Y (2016) Dynamic and automatic feedback-based thresh-
26 1473 old adaptation for code smell detection. *IEEE Transactions on Software Engi-*
27 1474 *neering* 42(6):544–558, DOI 10.1109/TSE.2015.2503740
28 1475 López Nozal C (2012) Design defects detection based on code metrics (in spanish).
29 1476 PhD thesis, Dpto. Informática, Universidad de Valladolid, URL [https://www.](https://www.educacion.gob.es/teseo/mostrarRef.do?ref=1003002)
30 1477 [educacion.gob.es/teseo/mostrarRef.do?ref=1003002](https://www.educacion.gob.es/teseo/mostrarRef.do?ref=1003002)
31 1478 Maiga A, Ali N, Bhattacharya N, Sabane A, Gueheneuc YG, Aimeur E (2012a)
32 1479 Smurf: A svm-based incremental anti-pattern detection approach. In: *Intl. Conf.*
33 1480 *on Reverse engineering*, pp 466–475
34 1481 Maiga A, Ali N, Bhattacharya N, Sabané A, Guéhéneuc YG, Antoniol G, Aimeur
35 1482 E (2012b) Support vector machines for anti-pattern detection. In: *Intl. Conf.*
36 1483 *Automated Software Engineering*, pp 278–281
37 1484 Maneerat N, Muenchaisri P (2011) Bad-smell prediction from software design
38 1485 model using machine learning techniques. In: *Intl. J. Conf. on Computer Science*
39 1486 *and Software Engineering*, pp 331–336
40 1487 Marinescu C, Marinescu R, Mihancea PF, Wettel R (2005) iPlasma: An integrated
41 1488 platform for quality assessment of object-oriented design. In: *Intl. Conf. Software*
42 1489 *Maintenance - Industrial and Tool Volume*, pp 77–80
43 1490 Matthews BW (1975) Comparison of the predicted and observed secondary struc-
44 1491 ture of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA)-Protein Struc-*
45 1492 *ture* 405(2):442–451
46 1493 Mihancea PF, Marinescu R (2005) Towards the optimization of automatic de-
47 1494 tection of design flaws in object-oriented software systems. In: *Ninth Euro-*
48 1495 *pean Conference on Software Maintenance and Reengineering*, pp 92–101, DOI
49 1496 10.1109/CSMR.2005.53
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- 1
2 1497 Mistrik I, Soley R, Ali N, Grundy J, Tekinerdogan B (eds) (2015) Software Quality
3 1498 Assurance: In Large Scale and Complex Software-intensive. Morgan Kaufmann
4 1499 Moha N (2007) Detection and correction of design defects in object-oriented de-
5 1500 signs. In: Conf. on Object-oriented Programming Systems and Applications
6 1501 companion, pp 949–950
7 1502 Moha N, Guéhéneuc YG (2007) DECOR: a tool for the detection of design defects.
8 1503 In: Intl. Conf. on Automated Software Engineering, pp 527–528
9 1504 Moha N, Gueheneuc YG, Duchien L, Le Meur AF (2010) Decor: A method for
10 1505 the specification and detection of code and design smells. IEEE Transactions on
11 1506 Software Engineering 36(1):20–36
12 1507 Morales R, Soh Z, Khomh F, Antoniol G, Chicano F (2017) On the use of develop-
13 1508 ers' context for automatic refactoring of software anti-patterns. J Systems and
14 1509 Software 128:236–251
15 1510 Mori A, Vale G, Vigiato M, Oliveira J, Figueiredo E, Cirilo E, Jamshidi P, Kastner
16 1511 C (2018) Evaluating domain-specific metric thresholds: An empirical study. In:
17 1512 Proceedings of the 2018 International Conference on Technical Debt, ACM,
18 1513 TechDebt '18, pp 41–50, DOI 10.1145/3194164.3194173
19 1514 Munro MJ (2005) Product metrics for automatic identification of "bad smell"
20 1515 design problems in java source-code. In: Intl. Conf. Software Metrics, pp 15–15
21 1516 Paiva T, Damasceno A, Figueiredo E, Sant'Anna C (2017) On the evaluation of
22 1517 code smells and detection tools. Journal of Software Engineering Research and
23 1518 Development 5(1):7
24 1519 Palomba F, Bavota G, Di Penta M, Oliveto R, De Lucia A (2014a) Do they really
25 1520 smell bad? a study on developers' perception of bad code smells. In: Intl. Conf.
26 1521 on Software maintenance and evolution, pp 101–110
27 1522 Palomba F, Lucia AD, Bavota G, Oliveto R (2014b) Anti-pattern detection: Meth-
28 1523 ods, challenges, and open issues. In: Memon A (ed) Advances in Computers,
29 1524 vol 95, Elsevier, chap 4, pp 201–238, DOI <https://doi.org/10.1016/B978-0-12-800160-8.00004-8>
30 1525
31 1526 Palomba F, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A (2015)
32 1527 Mining version histories for detecting code smells. IEEE Transactions on Soft-
33 1528 ware Engineering 41(5):462–489
34 1529 Pecorelli F, Palomba F, Di Nucci D, De Lucia A (2019) Comparing heuristic and
35 1530 machine learning approaches for metric-based code smell detection. In: Proceed-
36 1531 ings of the 27th International Conference on Program Comprehension, IEEE
37 1532 Press, ICPC '19, p 93–104, DOI 10.1109/ICPC.2019.00023
38 1533 Pecorelli F, Di Nucci D, De Roover C, De Lucia A (2020) A large empirical as-
39 1534 sessment of the role of data balancing in machine-learning-based code smell
40 1535 detection. Journal of Systems and Software 169, DOI [https://doi.org/10.1016/](https://doi.org/10.1016/j.jss.2020.110693)
41 1536 [j.jss.2020.110693](https://doi.org/10.1016/j.jss.2020.110693)
42 1537 Peiris M, Hill JH (2014) Towards detecting software performance anti-patterns
43 1538 using classification techniques. ACM SIGSOFT Software Engineering Notes
44 1539 39(1):1–4
45 1540 Powers D (2011) Evaluation: from precision, recall and f-measure to roc, informed-
46 1541 ness, markedness and correlation. Intl J of Mach Learning Technol 2(1):37–63
47 1542 Rapu D, Ducasse S, Gîrba T, Marinescu R (2004) Using history information to
48 1543 improve design flaws detection. In: Conf. on Software Maintenance and Reengi-
49 1544 neering, pp 223–232
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- 1
2 1545 Rasool G, Arshad Z (2015) A review of code smell mining techniques. *J Software: Evolution and Process* 27(11):867–895
- 3 1546
- 4 1547 Riel AJ (1996) *Object-oriented design heuristics*, vol 335. Addison-Wesley Reading
- 5 1548 Santos JA, de Mendonça MG, Silva CV (2013) An exploratory study to investigate the impact of conceptualization in god class detection. In: *Intl. Conf. on Evaluation and Assessment in Software Engineering*, pp 48–59
- 6 1549
- 7 1550
- 8 1551 Santos JAM, Rocha-Junior JB, Prates LCL, do Nascimento RS, Freitas MF, de Mendonça MG (2018) A systematic review on the code smell effect. *Journal of Systems and Software* 144:450 – 477
- 9 1552
- 10 1553
- 11 1554 Shatnawi R (2015) Deriving metrics thresholds using log transformation. *J Software: Evolution and Process* 27(2):95–113
- 12 1555
- 13 1556 Simons C, Singer J, White DR (2015) Search-based refactoring: Metrics are not enough. In: Barros M, Labiche Y (eds) *Search-Based Software Engineering*, Springer International Publishing, pp 47–61
- 14 1557
- 15 1558
- 16 1559 Tahvildar L, Kontogiannis K (2004) Improving design quality using meta-pattern transformations: a metric-based approach. *J Software: Evolution and Process* 16(4-5):331–361
- 17 1560
- 18 1561
- 19 1562 Tempero E, Anslow C, Dietrich J, Han T, Li J, Lumpe M, Melton H, Noble J (2010) The qualitas corpus: A curated collection of Java code for empirical studies. In: *Asia Pacific Software Engineering Conf.*, pp 336–345
- 20 1563
- 21 1564 Tsantalís N, Chaikalís T, Chatzigeorgiou A (2008) Jdeodorant: Identification and removal of type-checking bad smells. In: *Intl. Conf. on Software Maintenance and Reengineering*, pp 329–331
- 22 1565
- 23 1566
- 24 1567
- 25 1568 Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2015) When and why your code starts to smell bad. In: *Intl. Conf. on Software Engineering-Volume 1*, pp 403–414
- 26 1569
- 27 1570
- 28 1571 Vale G, Fernandes E, Figueiredo E (2019) On the proposal and evaluation of a benchmark-based threshold derivation method. *Software Quality Journal* 27(1):275–306, DOI 10.1007/s11219-018-9405-y
- 29 1572
- 30 1573
- 31 1574 Witten IH, Frank E, Hall MA, Pal CJ (2016) *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann
- 32 1575
- 33 1576 Yamashita A, Moonen L (2013) Exploring the impact of inter-smell relations on software maintainability: An empirical study. In: *Intl. Conf. on Software Engineering*, pp 682–691
- 34 1577
- 35 1578
- 36 1579 Yamashita A, Zanoni M, Fontana FA, Walter B (2015) Inter-smell relations in industrial and open source systems: A replication and comparative analysis. In: *Intl. Conf. on Software Maintenance and Evolution*, pp 121–130
- 37 1580
- 38 1581
- 39 1582 Zhang M, Hall T, Baddoo N (2011) Code bad smells: a review of current knowledge. *J of Software: Evolution and Process* 23(3):179–202
- 40 1583
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65