Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

# Prioritization of god class design smell: A multi-criteria based approach

Khalid Alkharabsheh[a], Sadi Alawadi[b,c,*], Karam Ignaim[a], Nabeel Zanoon[d], Yania Crespo[e], Esperanza Manso[a,e], José A. Taboada[f]

[a] Department of Software Engineering, Prince Abdullah bin Ghazi Faculty of Information and Communication Technology, Al-Balqa Applied University (BAU), Jordan
[b] Department of Information Technology, Uppsala University, 75105 Uppsala, Sweden
[c] Center for Applied Intelligent Systems Research, School of Information Technology, Halmstad University, 30118 Halmstad, Sweden
[d] Applied Science Department, Aqaba College, Al-Balqa Applied University (BAU), Jordan
[e] Departamento de Informática. Escuela de Ingeniería Informática. Campus Miguel Delibes, Universidad de Valladolid, Paseo de Belén 15, Valladolid 47011. Spain
[f] CiTIUS, Centro Singular de Investigación en Tecnoloxías Intelixentes, Universidad de Santiago de Compostela, Santiago de Compostela 15782. Spain

A R T I C L E  I N F O

*Article history:*
Received 19 May 2022
Revised 11 September 2022
Accepted 12 September 2022
Available online 21 September 2022

*Keywords:*
Software quality
Design smell
Design smell prioritization
Historical information
Empirical evaluation
God class

A B S T R A C T

**Context:** Design smell Prioritization is a significant activity that tunes the process of software quality enhancement and raises its life cycle. **Objective:** A multi-criteria merge strategy for Design Smell prioritization is described. The strategy is exemplified with the case of God Class Design Smell. **Method:** An empirical adjustment of the strategy is performed using a dataset of 24 open source projects. Empirical evaluation was conducted in order to check how is the top ranked God Classes obtained by the proposed technique compared against the top ranked God class according to the opinion of developers involved in each of the projects in the dataset. **Results:** Results of the evaluation show the strategy should be improved. Analysis of the differences between projects where respondents answer correlates with the strategy and those projects where there is no correlation should be done.

## 1. Introduction

Due to the large dimensions of software systems formed by millions of lines of code, many Design Smells (code smells, bad smells, architectural smells) could be identified. Automatic detection is useful, but a large number of detected smells is considered a problem. On the one hand, software modules differ in their stability, importance and criticality inside the system. Thus, not all detected smells in these modules will be repaired because they vary in their negative influence on the software quality (Brown et al., 1998). On the other hand, the required efforts to resolve the whole smell detected can outstrip the available human resources, time, and budget.

To assist developers in overcoming these limitations, the activity of prioritizing Design Smells is promising. Several approaches have been proposed for design smell prioritization (Singh et al., 2021; Verma et al., 2021; Kaur et al., 2021; Islam et al., 2022; Arcoverde et al., 2013; Ouni et al., 2015; Fontana et al., 2015; Marinescu, 2012), in which some of them shown in Section 2. Various factors were considered in those approaches to rank detected smells, such as detection reliability, historical information, developer context, severity, etc. According to these approaches and the factors intervening in prioritization, the obtained results showed different rankings for the same list of Design Smells.

In this work, we propose a design smell prioritization approach based on merging different combinations of criteria. The proposed approach has been empirically evaluated by professional developers, specifically in the case of God Class prioritization in the context of 24 open source software systems.

Section 2 presents related work and highlights the main differences with this proposal. Section 3 describes the criteria and parameters involved. An empirical adjustment of the multi-criteria merging approach is performed by focusing on the God Class Design Smell using a dataset obtained from 24 open source projects (Section 4). Section 5 presents the design, execution and analysis of the conducted empirical evaluation. Section 6 threats to validity discussion. Finally, conclusions and future lines of research are discussed in Section 7.

## 2. Related Work

A systematic literature review (Kaur et al., 2021; Alkharabsheh et al., 2018) published in 2018 and 2021, respectively, shows an

* Corresponding author.
*E-mail addresses:* khalidkh@bau.edu.jo (K. Alkharabsheh), sadi.alawadi@it.uu.se (S. Alawadi), karam.ignaim@bau.edu.jo (K. Ignaim), dr.nabeel@bau.edu.jo (N. Zanoon), yania@infor.uva.es (Y. Crespo), manso@infor.uva.es (E. Manso), joseangel.taboada@usc.es (J.A. Taboada).

https://doi.org/10.1016/j.jksuci.2022.09.011
1319-1578/

increase in the activity related to prioritization in recent years. This shows the importance of smell prioritization for adopting smell detection in software production.

Arcoverde et al. (2013) proposed four heuristics to rank smells based on their architectural relevance. The heuristics adapted different characteristics of a software component, in which two of them were related to the software history, namely, the number of modifications in the component and the number of errors detected in the component. Also, the number of smells present in the component, and finally, the main role that component plays in the software.

Ouni et al. (2015) proposed a meta-heuristic search approach for Design Smells correction tasks based on a prioritization strategy using four criteria. These criteria take into account the preferences of the developers in terms of prioritizing different kinds of Design Smell, the severity of Design Smells, the smell risk, and the importance of the smell concerning the whole project or concerning specific software components (packages or classes) which include the detected design Smells.

Fontana et al. (2015) proposed two kinds of Design Smell filters based on different strengths, namely, strong and weak, to remove the false positive smells or introduce indicators for the probability that they could be discarded. In the same year, 2015, also, Fontana et al. (2015) used a Design Smell Intensity index as a key to identify and prioritize the most critical smells. The intensity index is defined to capture the amount of smell associated with each Design Smell. The strategy focuses on exploiting the distribution of metrics and their threshold values to obtain different levels of intensity index for each detected Design Smell. The intensity of a smell is defined as an indicator of the smell severity (where the smell severity is measured by computing how many degrees the smell metric exceeds the specified threshold value Marinescu, 2012). Semantic descriptions illustrate the intensity level, which ranges from very low to very high. The proposed criteria are integrated into the JCodeOdor detection tool, which was developed by the authors. The criteria can also be integrated with fully automated detection tools by calculating the thresholds for the required metrics.

Vidal et al. (2016,) presented a semi-automated approach for prioritizing design smells, which they named SpIRIT. The proposed technique is based on a combination of three criteria: the stability of the software component in which the Design Smell is detected, an evaluation of each type of Design smell based on the developers perspectives, and modifiability scenarios for the software. In this approach, the developer perspective plays a leading role in the ranking process because developers can select the most critical Design Smell from their point of view by assigning values to smells based on an ordinal scale. Therefore, the developers choose the Design Smell that they prefer to deal with or the one they know about in terms of the negative effects it will cause to the system. Smell agglomerations are introduced and used to describe groups of smells related to each other (Oizumi et al., 2015). The relationship between smell agglomerations and an architectural concern and the behavior of smell agglomerations over the component history in terms of the number of smells in the agglomeration is also combined. The strategy is supported by the JSpIRIT tool.

Sae-Lim et al. (2016, 2017) presented a technique to prioritize smells that works by estimating the developer context by applying an impact analysis technique on textual information. Data on the context are collected through a list of issues obtained from an issue tracking system, including textual information related to each software module. Based on the results of the impact analysis, the detected smells are prioritized. Verma et al. (2021), highlighted the influence of various elements on code smell prioritization, such as detection tools, metrics, and factors. It's typical that when releases of object-oriented software increase, the ratio of code smell also increases. Understanding the drawbacks of code smell will improve the accuracy of the proposed approaches. Recently, Islam et al. (2022), proposed a novel approach for prioritizing code smells based on collecting information related to code parts that are frequently used and can change prone. The information is obtained from the comments history analysis, log files mining, and static code analysis. The proposed technique was evaluated using a case study of medium size project implemented in Java. The result shows the approach has a better performance compared with the same works regarding the developer-oriented testing phase to detect code smells.

Despite the diversity of the proposed prioritization approaches but there exist limitations concerning the selected criteria. For example, some works, such as Verma et al. (2021) and Arcoverde et al. (2013) did not consider the role of human context, while in other works (Ouni et al., 2015; Marinescu, 2012), the historical information dimension was discarded. Moreover, the number of parameters used to compute each criterion is another challenge where only one parameter has been used to measure the selected criterion in some works. Nevertheless, we assume that these criteria have a significant role in the activity of design smell prioritization and should be considered.

In our work, three criteria were selected to develop a technique for prioritizing smells based to be repaired. The criteria cover different aspects of software components, namely, historical information, smell density & intensity, and the assessment of developer context. The main differences between our approach and those of previous works are that we foster the use of the same quality tool of several smell detection tools as an expert committee. Several tools can detect the same smell, and this fact influences the smell intensity. As part of the historical information criterion, we consider the volume of changes and the significance of changes. Also, we have employed multiple weighting strategies to rank the smells to give different relative importance to the selected parameters of the criteria. The strategy was empirically adjusted using a dataset formed by different versions of 24 open source projects. We focus on a particular smell, God Class, for empirical adjustment and evaluation but define the strategy in a general way to be used for different smells. Finally, the approach was evaluated by different developers related to the same open source software project development. A web-based survey was designed to assess empirically the ranking list generated by our technique with the same project developers.

## 3. Proposed Approach

The proposed approach focused on the prioritization of God Class detection. Nevertheless, as mentioned before, the way the proposal is designed facilitates generalization to other smells. However, efforts in validation and adjustment to that other smells should be made. Defining priorities among different types of smells is beyond the scope of this work. The approach is organized in five steps, as shown in Fig. 1. The first three steps (Step1, Step2, and Step3) are focused on the dataset preparation, while Step4 is focused on computing the parameters of each criterion. Finally, in Step5, the list of ranked god classes is obtained.

**Step 1**. In this step, we randomly selected a set of software projects implemented in Java in which the source code is the input of the approach. The selected projects were of different sizes and domains. They were downloaded from the SourceForge repository, one of the most known repositories used in the open-source context.

**Step 2**. Here, the source code of the target version of the software is analyzed using multiple smell detectors. Several tools were standard in God Class detection. We selected a set of tools that

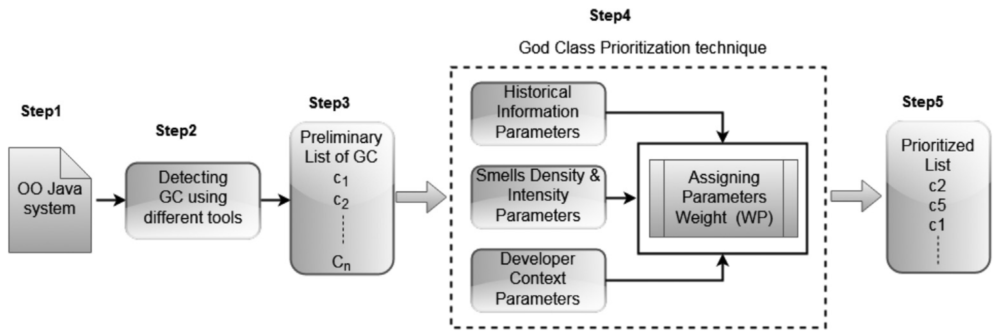

Step1
OO Java system
Step2
Detecting GC using different tools
Step3
Preliminary List of GC
c1
c2
Cn
Step4
God Class Prioritization technique
Historical Information Parameters
Smells Density & Intensity Parameters
Developer Context Parameters
Assigning Parameters Weight (WP)
Step5
Prioritized List
c2
c5
c1

**Fig. 1.** Prioritization technique.

were the most cited and used in the context. Moreover, have a high precision of god class detection.

**Step 3**. Once the software systems are analyzed in the previous step (Step2), in this step, the output of each tool will be a part of the preliminary list of god classes obtained from all tools (union set).

**Step 4**. Next, in Step5, which is considered the core of the proposed approach, some parameters are computed for each God Class in the union set for each of the three criteria. The examined aspects focused on the stability, maintainability, and the developers' context assessment. First, the stability of the classes on the target version is tackled through the criterion of historical information. For this purpose, a sample of earlier versions should be examined. Also, the maintainability aspect is evaluated by considering the density & intensity of smells in the target version. Finally, the developers' importance regarding God Class detection in the earlier version is considered. Once the set of parameters to measure each criterion is collected. Then, we try to assign the suitable weight for each parameter to obtain a rank as a weighted sum of all the selected parameters that allow classifying all God Classes in a prioritized list.

**Step 5**. Finally, in this step, we obtained the final prioritized list of god classes.

### *3.1. Detecting Design Smell*

The purpose of using several detection tools acting as a committee is to obtain different diagnoses on the same item. This approach leads to reducing the false negatives but can introduce false positives. Nevertheless, coincidences among tools indicate how "intense" is the presence of the smell in the item, leading to an agreement between tools. Therefore, as long as we focus on the empirical adjustment and validation in God Class prioritization, the rest of the text refers to the smelly items as "classes".

### *3.2. Computing Parameters for Historical Information Criterion*

This criterion focused on two issues related to class stability. On the one side, it is considered the volume of changes from a previous version to the target version. On the other hand, if the change occurred, we try to capture whether it has been significant or not. We use software change history also as a measure of code importance. Its variation reflects the interest and concern of developers to correct errors or increase its functionality. In our experience, if a class is frequently changed, we suspect similar behavior will be followed in the next versions. For this reason, we highlight the parameters of the class that have frequent and significant changes to prioritize repairing it. The following parameters have been selected: a variation on Number of Method (△NOM), a variation on Lines of Code (△LOC), and a variation in Cyclomatic Complexity (△CC) that we call SOC. SOC stands for Significance of Changes. The NOM and LOC are used to measure the volume of changes, while the CC is used for the significance of changes. NOM is related to the number of responsibilities and tasks the class will execute. Therefore, NOM increasing or decreasing will influence these responsibilities. On the other hand, if no changes occurred on the NOM inside the class, but the changes happened on the LOC inside the body of the method or the class attributes, the changes could be considered an indicator of some refactoring operation or bug repairing. Several empirical studies in the literature (Curtis et al., 1979; Jay et al., 2009; Jbara et al., 2014; Landman et al., 2014; Tashtoush et al., 2014) showed there is a significant correlation between cyclomatic complexity and lines of code. Intuitively, the larger the lines of code (LOC) or the number of methods (NOM), the more branches, the higher the cyclomatic complexity. Therefore, we can reasonably expect that increasing or decreasing NOM or LOC will have an effect on the cyclomatic complexity of the class under analysis. Nevertheless, when variation NOM and/ or LOC do not lead to variation in cyclomatic complexity, the change will be considered without significance and penalized in the weighted sum.

In order to unify the scale of parameter values, the results are normalized by computing the relative measure of all parameters against the maximum value of the previous or target class. In the following paragraphs, we described in detail the selected parameters.

△**Number of Methods (NOM).** Starting from the selected versions [previous version (i), target version (j)] of the project, △NOM = $—NOM_i$-$NOM_j—$ is calculated. The obtained result of △NOM ranged from 0 to N, where △NOM = 0 if no changes happened (same number of methods/ responsibilities from one version to the other). To normalize the results, we compute the relative measure (RelNOM) of a particular class (c) as follows:

$$RelNOM_{(c)} = \frac{\Delta NOM}{max(NOM_i, NOM_j)}, \quad NOM_i, NOM_j > 0$$

△**Lines of Code (LOC).** In the same way, variation in LOC is calculated △LOC = $—LOC_i$-$LOC_j—$. To obtain a normalized value, the relative measure (RelLOC) of changes in LOC of a particular class (c) is computed as follows:

$$RelLOC_{(c)} = \frac{\Delta LOC}{max(LOC_i, LOC_j)}, \quad LOC_i, LOC_j > 0$$

**Significance of Changing (SOC).** To compute the significance of changing (SOC) of a particular class (c) between any two versions (i, j) of a project, firstly, variation △SOC = $—CC_i$-$CC_j—$ is obtained. Then, the relative measure of (RelSOC) is computed as follows:

$$RelSOC_{(c)} = \frac{\Delta SOC}{max(SOC_i, SOC_j)}, \quad SOC_i, SOC_j > 0$$

These three parameters can be calculated automatically and integrated with any tool.

### 3.3. Computing Parameters for Smell Density & Intensity Criterion

Different studies have shown a lack of agreement in detecting smells among various detection tools. With the criterion of intensity, which assesses the number of tools that detect the smell in the class, we highlight among the detected those with the most significant possibility of being truly positive. In addition, the relationship between the presence of the aimed smell in the class and other types of smells complement "intensity" with "density". Density is focused on determining the volume of problems found in the class by identifying different types of smells and their amount of total repetitions. The impact of code changes on the smell density along the different versions of the same class is addressed. A class having a high smell density will be harmful and should be revised and improved first.

To this end, we selected different parameters to represent the criterion of Density & Intensity. The chosen parameters are explained in detail In the following paragraphs:

**Number of Detection Tools (DT).** This parameter is concerned with computing the number of tools that detect the smell in a particular class (c). The number of tools is different based on the type of smell we want to detect and the availability of different tools that can detect it. The parameter value is considered an indicator of the agreement of detection tools. Moreover, the higher value, the more intensity in the presence of the Design Smell. It is interpreted as it is straightforward for several "experts" that the class is smelly.

A set of factors affect the integration of this parameter to the top of other tools. Some tools are designed to work standalone, by means of plug-ins, or both. Moreover, the implementation of tools determines how the tool should be used, either through a graphical interface or by commands or both. Therefore, depending on the tool, the value of this parameter can be obtained fully automated or semi-automatically according to the current version of some detection tools. The parameter value is computed as follows:

$$DT_{(c)} = \sum_{i=1}^{n} T^{i}_{(c)}, \quad T^{i}_{(c)} \in \{0,1\}$$

DT$_{(c)}$ is the number of tools that detect the particular smell in class (c), $n$ is the number of tools and $T^i_c$ is the tool (i) that detects the smell in class (c). This value is 1 if detected or 0 if not. $DT_{(c)}$ range from 1 to $n$. So, it should be necessary to be normalized. Relative measure (RelDT) is calculated as:

$$RelDT_{(c)} = \frac{DT_{(c)}}{n}, \quad n > 0$$

**Design Smell Types (DST).** This parameter calculates the number of the different smell types found in the class according to the set of available detection tools. Whenever different types of smells are detected in the class, the number of quality factors that are negatively affected increases, and therefore the number of various issues that the developers will need to focus on to fix. As a consequence, if the class is not immediately repaired, the future maintenance effort will increase. The importance of this parameter lies in identifying the group of smells that are related to each other, what is called in the literature "Design Smells Agglomeration" as mentioned in Section 2 (Oizumi et al., 2016; Oizumi et al., 2017; Oizumi et al., 2015; Carvalho and Mendonça, 2018; da Silva Sousa, 2016), where the presence of a particular smell indicates the presence of other types of smells in the same class, the parameter's value depends on the number of smells each tool is designed to detect. In this parameter, we considered the smell type detected by more than one tool as different due to the ambiguities in the definition of the smell used by each tool and the applied detection strategy. The parameter is calculated as follows:

$$DST_{(c)} = \sum_{i=1}^{n} DST^{i}_{(c)}, \quad DST^{i}_{(c)} \in \{0,1\}$$

Where (DST$_{(c)}$) is the number of different types of Design Smells that are detected in a particular class, $n$ is the number of Design Smell types and (DST$_i$) is the Design Smell type ($i$) that is detected by the tools, in which the value is 1 if detected or 0 if not. The scale of this parameter is different from one class to another; thus, it should be normalized by obtaining the relative measure of the Design Smells types (RelDST) as follows:

$$RelDST_{(c)} = \frac{DST_{(c)}}{n}, \quad n > 0$$

**Total repetitions of all types of Design Smells (TotDST).** Using this parameter, the number of problems (Design Smells) that could be found in the class are identified. In this parameter, we will compute the summation of the total repetitions of all types of Design Smells detected in the class using the available detection tools. The parameter calculation is defined as follows:

$$TotDST_{(c)} = \sum_{i=1}^{n} F^{i}_{(c)}, \quad F^{i}_{(c)} \in \{\mathbb{N}\}, \quad n \geqslant 0$$

Where (TotDST) the total repetitions of all Design Smell types that are detected in the particular class (c), ($F^i_{(c)}$) factor represents how many times the Design Smell type ($i$) appears in class c (i.e. the repetition), ($i$) represents each Design Smell type detected, and ($n$) the number of different Design Smells types detected by all the selected tools. The obtained values for different classes will be various in their ranges. So, it is necessary to adjust the values to the same scale as the other parameters. For this purpose, as a first step, we computed the ratio (TotDST):

$$RatioTotDST_{(c)} = \frac{TotDST_{(c)}}{DST_{(c)}}, \quad DST_{(c)} > 0$$

Then, we use the Min–Max normalization technique (Saranya and Manikandan, 2013) in order to adjust the result of RatioTotDST$_{(c)}$ for each class between 0 and 1 as follows:

$$NormalizedRatioTotDST_{(c)} = \frac{RatioTotDST_{(c)} - Min(RatioTotDST)}{Max(RatioTotDST) - Min(RatioTotDST)}$$

### 3.4. Computing Parameters for Developers Context Criterion

Several studies in the state of the art (Arcoverde et al., 2011; Palomba et al., 2014; Peters and Zaidman, 2012; Sae-Lim et al., 2016; Sae-Lim et al., 2017; Sae-Lim et al., 2017; Yamashita and Moonen, 2012; Alkharabsheh et al., 2016; Alkharabsheh et al., 2019) have focused on the importance of developers' context regarding their years of experience, background, awareness of Design Smells, etc., on Design Smell evaluation. In our experience, detection approaches that ignore developers' context are not fully satisfactory because these approaches do not consider the subjectivity related to the persons on smell detection. Thus, the results of these approaches could be Design Smells that are not relevant or significant to the developers. The role of professional developers is to approve or reject Design Smells detected by automatic tools, especially in large-scale software systems when the results of detection tools include an extensive list of Design Smells. Modern quality assurance tools such SonarQube include in their functionality the possibility that the developers or QAs indicate false positives.

The developers assessment (DevEvl) parameter is a binary decision (1,0), in which the result will be (1) when the developers confirm the smell detected in class (c) that has been detected by tools. Otherwise (0), when the developer rejects that (c) is smelly. The default value of this parameter is (0) because all the studies in the literature indicate that there is a lack of agreement between humans, tools and both.

$$DevEvl_{(c)} = \begin{cases} 1, & \textit{If the developer confirms true positive} \\ 0, & \textit{Otherwise} \end{cases}$$

## 4. Empirical Adjustment of the Multi-criteria Approach

As mentioned before, the case of God Class Design Smell is focused on tackling empirical adjustment and empirical evaluation of the proposed prioritization technique. The whole parameters are going to be combined and adjusted in a way that the prioritization technique can be able to meet the final purpose of the study. Whether the selected criteria and parameters used to prioritize the detected God Classes were relevant to the ranking that developers believe. To detect God Class, there exists a long list of prototypes and tools. We avoided the prototypes and concentrated on the automatic tools. We assigned a set of criteria to select the tools that include available and free, common in god class detection, analyzing java source code and must have a high accuracy in detection. From the obtained list, we selected a set of five tools that involve DÉCOR, iPlasma, PMD, Together, and JDeodorant. According to our systematic mapping study on design smell detection published in 2019 Alkharabsheh et al. (2018), the chosen tools were the most mentioned in the works related to the activity of design smell detection. The selected tools have used different strategies to detect the god class. The strategies were built based on the precise definitions of god class. The variation of detection strategies will increase the total number of god classes in the dataset and reduce the threats to construct validity. For example, JDeodorant employed the hierarchical agglomerative clustering algorithm, DECOR used the rule-based, and the group of iPlasma, PMD, and Together used a metric-based strategy, in which different sets of metrics and threshold values were used in the detection technique. They constitute the committee. In the next subsections, we start by describing the dataset built to be used for empirically setting up the combination of the previously presented parameters in Section 4.1. Then, in Section 4.2, we explain by example how this dataset is used in tuning the rankings, i.e., how we combine the selected parameters to produce the relevant God Class list for each software project.

### *4.1. Dataset*

The dataset is collected from 24 open source Java software projects (see Table 1) and used in several previous studies (Alkharabsheh et al., 2016; Alkharabsheh et al., 2016; Alkharabsheh et al., 2022; Alkharabsheh et al., 2021; Alkharabsheh et al., 2021; Alkharabsheh, 2021) for different purposes.These software projects range from small to large sizes and belong to different domains. The dataset used in this study is available on the web[1]. Two different versions of each project were chosen, the target version and the previous version. The last column of the table presents the number of classes added to (+) or removed from (-) the previous version of the software projects. This number can be an indicator of the code improvement concerning eliminating classes or dividing the complex or large classes (god class) into two classes or more, which finally affects the total number of classes.

God Class detection was performed in both versions of the projects. Results show that the total number of God Classes in the previous versions was $1,958$, representing $15.6\%$ against the total number of classes of all projects in this version. The largest number of God Classes was detected in the Jasperreport v4.7.1 project (311 God Classes), and the lowest number was detected in the Plugfy v0.3 (1 God Class). On the other hand, considering together all the classes of the 24 projects, the total number of God Classes in the target versions was $1,131$, representing $9.11\%$ of the total. As can be seen, the number of god classes in the target version decreased compared with the number in the previous version (from 1,958 to 1,131). This decrease is expected because some refactoring operations have been conducted on the god classes detected in the previous versions, or some of these classes have been removed from the next versions of the software project.

The complete list of god classes in both versions can be accessed on the web[2] According to the numbers and rates of God Classes in both versions, we can conclude the problem was reduced in some projects such as GanttProject and JFreeChart in the target versions against the previous versions ($15.6\%$ to $9.11\%$). It could be an indicator of the code improvement (refactoring operations), specifically when the same class was identified in the previous versions as a God Class and the target version was not.

### *4.2. Constructing the Concrete God Class Ranked List*

The general strategy proposed is based on giving a Top X priority list to developers or QAs, where X is small enough to allow attention to be focused on those priorities and not too overwhelmed with many issues to solve. Whenever the developer or QA requests the priority list of God Classes, the Top X, the first X classes in order of priority, is given. Once the developer has already analyzed them and either created tasks to repair the presence of the smell, discarded reparation at this point, or indicated that he/she thinks it is not God Class (indicates false positive), then the exact mechanism is applied to get the next Top X.

It has been decided to instantiate X with 5 to exemplify the strategy and explain the construction of a Top 5. The reason to instantiate concretely in 5 is to directly use the Top 5 obtained for each project in the dataset in the stage of empirical validation that will be presented in the following section. In this empirical validation, these Top 5 will be given to developers involved in each project as a prioritized list of 5 elements to contrast their opinion on the ranking. Therefore all the strategy that will be explained below is talking about Top 5 but could be talking about Top 10, Top X in general, always thinking that X should be small enough to allow developers or QAs to focus the attention.

To understand how we will be constructed the concrete God Class rank (Top 5) for each software project, we present a running example of a software project from the dataset, JFreeChart. After analyzing the project's source code of the selected versions (v1.0.x, v1.5.0), our findings showed that 161 God Classes were detected in the previous version (v1.0.x). In contrast, 72 God Classes were detected in the target version (v1.5.0). A set of 72 God Classes was common among the project versions, which are the same god classes in the target version v1.5.0. that included in the preliminary list. In another example, FullSync, the number of god classes in v0.10.2 (previous version) was 13, while in v0.10.4 (target version) was 8. The common god classes between both versions were 3. As mentioned before, this study focused on the set of god classes common in both versions. Table 2 presents the number of god classes in the previous and target versions of the software projects,and the number of common God Classes in both versions.

[1] shorturl.at/DFLN4

[2] shorturl.at/DFLN4

**Table 1**
Characterization of the selected versions of projects.

| Name | Previous Version | | | target Version | | | # of Classes |
|---|---|---|---|---|---|---|---|
| | Version | NOC | TLOC | Version | NOC | TLOC | Inc(+)/Dec(-) |
| AngryIPScanner | 3.0 | 177 | 10,456 | 3.5 | 213 | 11,205 | +233 |
| Apeiron | 2.92 | 62 | 8,908 | 2.94 | 64 | 6,557 | +2 |
| Checkstyle | 6.2.0 | 116 | 7,039 | 8.0.0 | 169 | 10,546 | +53 |
| DigiExtractor | 2.3.1 | 77 | 610 | 2.5.2 | 80 | 15,668 | +3 |
| Freemind | 1.0.1 | 501 | 60,972 | 1.1.0 | 517 | 60,151 | +16 |
| FullSync | 0.10.2 | 211 | 13,915 | 0.10.4 | 200 | 13,536 | −11 |
| GanttProject | 2.0.10 | 621 | 66,540 | 2.8.8 | 652 | 39,798 | +31 |
| jasperreports | 4.7.1 | 1,797 | 350,690 | 6.2.2.x | 2,889 | 210,480 | +1092 |
| jAudio | 1.0.4 | 461 | 57,144 | 1.1.1 | 477 | 59,026 | +16 |
| Java graphplan | 1.0.7 | 56 | 6,481 | 1.0.12 | 56 | 6,512 | 0 |
| JCLEC | 4.0.0 | 305 | 13,556 | 4.0.x | 305 | 13,556 | 0 |
| JDistLib | 0.3.5 | 88 | 29,845 | 0.3.8 | 78 | 32,081 | −10 |
| JFreeChart | 1.0.x | 499 | 206,559 | 1.5.0 | 980 | 123,120 | + 481 |
| JHotDraw | 5.2 | 171 | 8,162 | 7.6 | 609 | 71,222 | +438 |
| keystore-explorer | 5.1 | 384 | 83,144 | 5.3.2 | 374 | 47,158 | −10 |
| Lucene | 3.0.0 | 393 | 42,351 | 6.5.0 | 1,392 | 96,66 | +999 |
| Matte | 1.7 | 603 | 52,067 | 1.8.2 | 659 | 40,037 | + 56 |
| Mpxj | 4.7.0 | 513 | 85,065 | 7.0.2 | 644 | 98,681 | + 131 |
| OmegaT | 3.1.8 | 629 | 58,348 | 4.1.1 | 932 | 56,557 | +303 |
| Plugfy | 0.3 | 50 | 1,039 | 0.6 | 52 | 1,089 | +2 |
| PMD | 4.3.x | 757 | 52,576 | 6.0.0 | 1,288 | 37,031 | +531 |
| Smeta | 0.9.1 | 229 | 14,241 | 1.0.3 | 222 | 30,843 | −7 |
| squirrel-sq | 3.7.1 | 1,138 | 71,626 | 3.8.1 | 1,598 | 129,711 | + 460 |
| Xena | 5.0.0 | 2,572 | 402,506 | 6.1.0 | 1,975 | 61,526 | −597 |

**Table 2**
Number of god classes in the previous and target version of each project.

| Project Name | Previous Version | | target Version | | # Common GodClass |
|---|---|---|---|---|---|
| | Version | # GodClass | Version | # GodClass | |
| AngryIPScanner | 3.0 | 2 | 3.5 | 4 | 2 |
| Apeiron | 2.92 | 16 | 2.94 | 9 | 8 |
| Checkstyle | 6.2.0 | 20 | 8.0.0 | 9 | 3 |
| DigiExtractor | 2.3.1 | 27 | 2.5.2 | 36 | 27 |
| Freemind | 1.0.1 | 42 | 1.1.0 | 62 | 32 |
| FullSync | 0.10.2 | 13 | 0.10.4 | 8 | 3 |
| GanttProject | 2.0.10 | 167 | 2.8.8 | 33 | 33 |
| jasperreports | 4.7.1 | 311 | 6.2.2.x | 200 | 179 |
| jAudio | 1.0.4 | 121 | 1.1.1 | 78 | 76 |
| Java graphplan | 1.0.7 | 18 | 1.0.12 | 8 | 8 |
| JCLEC | 4.0.0 | 88 | 4.0.x | 88 | 88 |
| JDistLib | 0.3.5 | 22 | 0.3.8 | 17 | 17 |
| JFreeChart | 1.0.x | 161 | 1.5.0 | 72 | 72 |
| JHotDraw | 5.2 | 26 | 7.6 | 50 | 1 |
| keystore-explorer | 5.1 | 47 | 5.3.2 | 40 | 40 |
| Lucene | 3.0.0 | 136 | 6.5.0 | 70 | 14 |
| Matte | 1.7 | 31 | 1.8.2 | 25 | 25 |
| Mpxj | 4.7.0 | 127 | 7.0.2 | 111 | 85 |
| OmegaT | 3.1.8 | 174 | 4.1.1 | 52 | 41 |
| Plugfy | 0.3 | 1 | 0.6 | 0 | 0 |
| PMD | 4.3.x | 29 | 6.0.0 | 38 | 7 |
| Smeta | 0.9.1 | 20 | 1.0.3 | 16 | 16 |
| squirrel-sq | 3.7.1 | 79 | 3.8.1 | 55 | 55 |
| Xena | 5.0.0 | 217 | 6.1.0 | 50 | 50 |

For each God Class, we computed the parameter values associated with each criterion in the previous section. Therefore, these values are not shown here for short.

Developers' assessment of God Classes as true positives or false positives was obtained as part of previous work by means of a web survey. Modern QA tools such as SonarQube include functionality to allow developers or QAs to state whether they consider a result a false positive.

A weighting strategy is applied. The strategy only includes the set of parameters in the historical information (RelNOM, RelLOC, RelSOC) and Design Smell density criteria (RelDT, RelDST, RatioTotDST) because these parameters are from the same type of scale "numeric." All values are between 0 and 1, while the developer context is "Boolean," The values are either 0 or 1. Therefore, it is challenging to combine different types of variables. For these 6 numerical parameters, we present 3 possible scenarios in which the weight of the decision is distributed with different criteria.

**Scenario 1:** giving the same weight value to both criteria, having all parameters the same percentage. As mentioned above, the total weight is 100, and we have 6 parameters. Therefore, the percentage value for each parameter is $1/6 = 16.67\%$.

**Scenario 2:** giving the highest weight value to the Smell Density & Intensity criterion, where all parameters in the criterion shared the weight value in a similar percentage, while the remained weight value is divided between all parameters in the other criteria in equal percentage. Hence, we gave 66.7%(2/3) of the weight percentage to the smell density parameters (RelDT, RelDST, RatioTotDST), in which each parameter of the three has 22.2%, and the rest weight percentage (33.3%) distributed between the parameters of the other criterion (RelNOM, RelLOC, RelSOC), 11.11% for each parameter.

**Scenario 3:** giving the highest weight value to the historical information criterion, where all parameters in the criterion shared the weight value in a similar percentage, while the remained weight value is divided between all parameters in the other criteria in equal percentage. Therefore, we gave 66.6% of the weight percentage to the historical information parameters (RelNOM, RelLOC, RelSOC), in which each parameter has 22.2%, and the rest parameters of the other criterion (RelDT, RelDST, RatioTotDST) have 11.11% for each.

According to these scenarios, Table 3 shows the obtained weighted scores and the ranks (R) of the God Classes in the target version of the JFreeChart v1.5.0 project. To compute the position of the rank of each God Class in the list, we ranked them in descending order based on their weight scores. The God Class with the highest weight score is in the first position in the ranking. According to this mechanism, multiple ranking lists have been obtained to obtain a unified list from the previous ranks (R1, R2, R3). The next steps are followed for each project in the dataset.

**Step 1.** Obtaining the list of classes confirmed as a God Class according to the criterion of developers context can be seen in the motivating example in step 1 of Fig. 2. Even though all classes have been identified as God Classes by the automatic tools mentioned previously, the most critical is the set that has been confirmed by the developers as a God Class. If the list of God Classes in this step is empty, i.e., no God Class has been approved by the developers, then the list of the top five God Classes will be obtained directly from step 2 by choosing the first five classes have the highest weighting values.

**Step 2.** Obtaining the list of top ten God Classes based on each weighting scenario, so we have three lists (scenario1, scenario2, scenario3). Then, the duplicated classes between the three lists are merged into one class that has the highest weight value. After that, we combine the three lists (scenario1, scenario2, scenario3) into one list and sort it according to the weight values in descending order (step 2 in Fig. 2). If the list of God Classes after merging the three lists does not include common classes with the list in step 1 or does not have enough to obtain the top five classes, then the next ten God Classes should be obtained by the same process. The process continues until we get the top five list.

**Step 3.** Select the top five God Classes common between the two lists in steps 1 and 2, as shown in step 3 of Fig. 2.

The purpose of this strategy is to top rank classes that will be high in the list of priorities, whatever the weighting was. This is a multi-criteria merge.

## 5. Empirical Evaluation

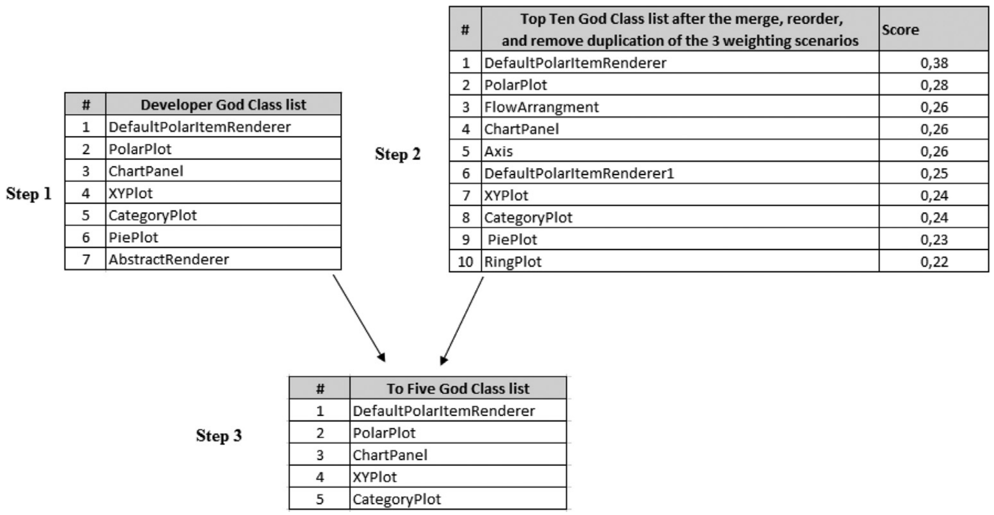
In order to evaluate the proposed technique, focusing on the case of God Class, we designed a web-based survey questionnaire to define priorities among the classes of the target version of software systems to decide which class to be repaired first by human experts, who are part of the project team or are regular or sporadic contributors.

**Research Question:** How are the top-ranked God Classes obtained by the proposed technique compared to the top-ranked God class according to the opinion of developers involved in the project?.

**Survey Design:** The survey questionnaire was constructed in three parts. In the first part, three questions related to the subject's

**Table 3**
The obtained scores and ranks for God Classes in the target version of JFreeChart v1.5.0 project according to each weighting scenario.

| Class Name | scenario1 | R1 | scenario2 | R2 | scenario3 | R3 |
|---|---|---|---|---|---|---|
| Axis | 0.24 | 3 | 0.26 | 2 | 0.22 | 3 |
| ChartPanel | 0.23 | 4 | 0.26 | 2 | 0.20 | 4 |
| DefaultPolarItemRenderer | 0.31 | 1 | 0.25 | 3 | 0.38 | 1 |
| PolarPlot | 0.26 | 2 | 0.27 | 1 | 0.28 | 2 |
| RingPlot | 0.22 | 5 | 0.21 | 4 | 0.22 | 3 |



| # | Developer God Class list |
|---|---|
| 1 | DefaultPolarItemRenderer |
| 2 | PolarPlot |
| 3 | ChartPanel |
| 4 | XYPlot |
| 5 | CategoryPlot |
| 6 | PiePlot |
| 7 | AbstractRenderer |

| # | Top Ten God Class list after the merge, reorder, and remove duplication of the 3 weighting scenarios | Score |
|---|---|---|
| 1 | DefaultPolarItemRenderer | 0,38 |
| 2 | PolarPlot | 0,28 |
| 3 | FlowArrangment | 0,26 |
| 4 | ChartPanel | 0,26 |
| 5 | Axis | 0,26 |
| 6 | DefaultPolarItemRenderer1 | 0,25 |
| 7 | XYPlot | 0,24 |
| 8 | CategoryPlot | 0,24 |
| 9 | PiePlot | 0,23 |
| 10 | RingPlot | 0,22 |

| # | To Five God Class list |
|---|---|
| 1 | DefaultPolarItemRenderer |
| 2 | PolarPlot |
| 3 | ChartPanel |
| 4 | XYPlot |
| 5 | CategoryPlot |

**Fig. 2.** List of top five God Classes in JFreeChart v1.5.0 project.

profile regarding their years of experience in software development, software project, and their role in the project. The second part involves the list of top five God Classes in the software project that should be repaired from first place to last according to the proposed technique. We decided to rank only the top five God Classes in each software project because it would not be acceptable to give the respondent an extensive list of God Classes requiring more time to analyze. The respondents are asked to either agree or propose another ranked list in case they disagree. Finally, in the last part, we asked the respondents to select the set of factors that have been taken into account when ranking the God Classes based on their importance. This last part was conducted as a replication of Sae-Lim et al. study (Sae-Lim et al., 2017) to answer another critical research question, but this part is beyond the scope of this paper. (To show the survey questions see reference K. et al., 2022).

**Execution:** To obtain responses from the available software project team and the contributors, they were contacted via the available addresses in the public repositories (GitHub, SourceForge) where the software projects were found. We sent a message to explain the situation and a link to the survey. The survey was duplicated 24 times and sent separately for each project to give each survey the top five God Class ranked list specific to each project. To facilitate the task of the experts, we provided a link in the survey to remember the definition of God Class Design Smell. The complete questionnaire is available on this url. The total number of contacted people via SourceForge was 49, while on GitHub was 89. In some cases, some people may be in both repositories because most of the projects migrated from SourceForge to GitHub. See the dataset on the web to show the number of contacted people in each project. Several criteria are considered during the survey design stage to choose the respondents. These criteria focused on the experience in software development, experience with the project, and the leading role in the project. We aim to increase the number of respondents who have a strong background as can as possible.

The survey stayed open for three months, from August 2018 until November 2018. We received feedback on 15 software projects in a ratio of 62.5% of the total number of projects. The respondents have extensive working backgrounds in software development and in the same software project they assess in this survey. Moreover, they have many years of experience. The final number of respondents for these projects was 35, and all answers were accepted. In the following, we provide more details about analyzing the respondents feedback.

### 5.1. Survey Analysis

**Subject's profile:** Fig. 3 presents the results related to the respondent profiles (the first part of the survey) regarding their years of experience and their leading role in the software project. Most respondents (54%) have more than ten years of experience, and 63% of the respondents have more than five years of experience in the same projects. As seen from both panels of the figure, most of them (37%) contributed to the project as software developers, while 3% were software testers. Also, more than 50% of the respondents acting as software developers in the project have more than ten years of expertise in software development and more than five years in the same project. Summarizing, more than 50% of respondents have enough expertise and knowledge on the project which satisfy the goals of the study.

**Top Five God Classes:** In this part of the survey, we asked the respondents to determine if they agreed or disagreed with our God Class prioritized list ranked as the top five of each software project. Then, based on their decision, the survey asks either to assign ranking positions to the existing list or to propose a new ranked list. Table 4 shows the distribution of the respondents answers related to their decision of agreement or disagreement with the top five God Classes listed by each software project. As shown, 48.6% (17 out of 35) of the respondents answers agree with our list, but with different ranking positions of the God Classes.

On the other hand, 51.4% (18 out of 35) of respondents disagreed with our list, in which 8 did not answer the question related to proposing another ranking list. In contrast, the rest of the respondents (10) suggested another ranked list that included God Classes from the lists that were obtained by using our technique. The presented ranking lists were the software projects

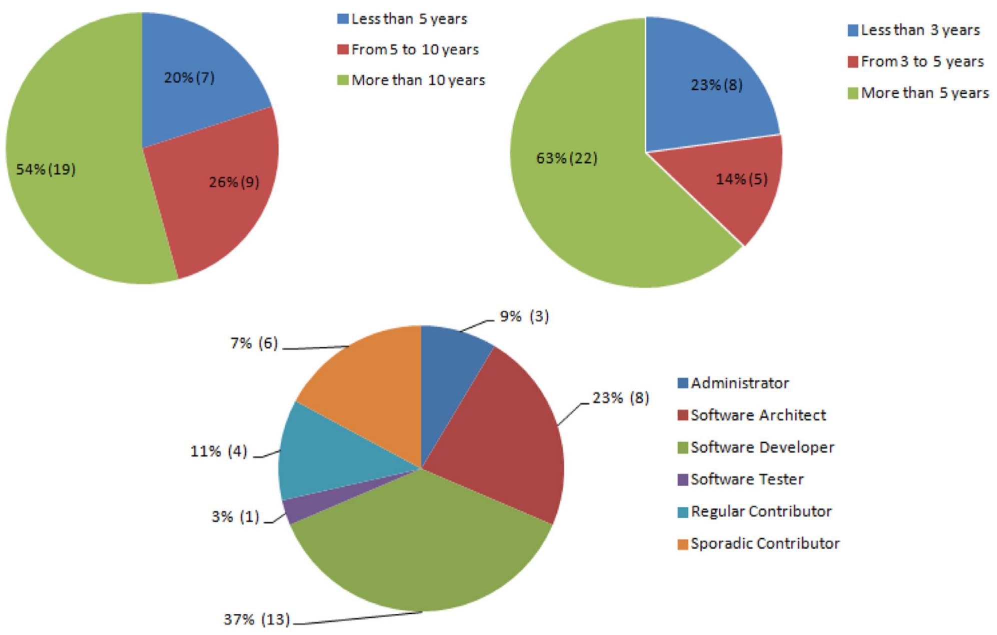


**Fig. 3.** Left up panel: Distribution of percentage and numbers of respondents over the expertise years in software development. Right up panel: Distribution of percentage and numbers of respondents over the expertise years in the same software project. Center down panel: Distribution of percentage and numbers of respondents over the roles that played in the software project.

**Table 4**
Distribution of respondents numbers over the selected software projects with respect to who answered the survey, who agree with our top five list, and who disagree with our top five list.

| Project Name | Version | # Respondents | Agree | Disagree |
|---|---|---|---|---|
| Angry IPScanner | 3.5 | 2 | 1 | 1 |
| checkstyle | 8.0.0 | 2 | 1 | 1 |
| Freemind | 1.1.0 | 1 | 1 | 0 |
| FullSync | 0.10.4 | 1 | 0 | 1 |
| Ganttproject | 2.8.8 | 3 | 3 | 0 |
| JAudio | 1.1.1 | 2 | 1 | 1 |
| Java graphplan | 1.0.12 | 1 | 1 | 0 |
| JCLEC | 4.0.0 | 2 | 0 | 2 |
| JFreeChart | 1.5.0 | 3 | 2 | 1 |
| Jhotdraw | 7.6 | 3 | 1 | 2 |
| Lucene | 6.5.0 | 5 | 1 | 4 |
| MPXJ | 7.0.2 | 1 | 0 | 1 |
| OmegaT | 4.1.1 | 2 | 1 | 1 |
| Plugfy | 0.6 | 2 | 1 | 1 |
| PMD | 6.0.0 | 5 | 3 | 2 |
| Total | | 35 | 17 | 18 |
| Percentage | | | 48.6% | 51.4% |

Checkstyle-8.0.0, FullSync-0.10.4, Jhotdraw-7.6, Lucene-6.5.0, JCLEC-4.0.0 and PMD-6.0.0. In these projects, 21 new God Classes were added by the respondents to be in the top five list, while 19 God Classes were selected from our top five list.

The highest number of respondents was 5 in two cases of the projects (Lucene-6.5.0, PMD-6.0.0). In addition, in 5 software projects, the number of respondents who agree with our technique equals the number of respondents who disagree.

To identify to what extent the similarity of ranking positions between our technique and respondents answers was, we hypothesised that there was a correlation between the ranking positions of our technique and the respondents answers. For this purpose, we computed Spearman's correlation coefficient (Pirie, 2004) using the R tool. Spearman's correlation value can range from $-1$ to 1. The higher the value of correlation, the stronger similarity. The perfect similarity is when the correlation is 1 while $-1$ implies a perfect inverse correlation.

We computed the correlation with different categories of respondents according to their decision if they agree or disagree with the technique, their role in the project, and the type of experience as shown in Tables 6–8. In general, a weak correlation result was found between our technique and the respondents ranking.

Table 5 shows the interpretation of this coefficient.

Table 6 presents the correlation between our technique and the categories of all respondents, the group of respondents who agree with our approach, and the group that disagrees but includes God Classes from our top five regarding the ranking positions. As can be seen, there is a correlation between both rankings, where there was a very weak correlation (0.12) in the case of all respondents, and only a weak correlation (0.28) was found in the case of respondents who agree with us. Despite the correlation is weak, we decided to make more analyses with this group of respondents to identify if there is a relationship between the role of respondents in the software project or the experience type with their ranking decisions.

**Table 5**
Interpretation of the Spearman's correlation values.

| Spearman's correlation value | Degree of Correlation |
|---|---|
| $0.00 \leqslant Corr < 0.19$ | Very weak |
| $0.20 \leqslant Corr < 0.39$ | Weak |
| $0.40 \leqslant Corr < 0.59$ | Moderate |
| $0.60 \leqslant Corr < 0.79$ | Strong |
| $0.80 \leqslant Corr \leqslant 1.00$ | Very strong |

**Table 6**
Correlation values between our prioritization technique and the different categories of respondents that include all, agree, and disagree (group that were include God Class from our list) regarding the ranking positions.

| Respondents Category | All | Agree | Disagree |
|---|---|---|---|
| # Respondent | 35 | 17 | 3 |
| Correlation | 0.12 | 0.28 | $-0.004$ |

Table 7 shows the correlation between both rankings (technique and the respondents who agree) based on classifying the respondent into small groups according to their role in the project. Even though there was a small number of respondents in each category, we found a correlation value ranging from strong to very strong in two cases when the role of respondents was Software Tester or Regular Contributor in the project. Despite that the developers category is the closest to the nature of the code, a low correlation is found. In our opinion, the reason is related to the developers experience and the team's instability over time. Therefore, we compute the correlation only with the group of developers with the highest years of experience (5 developers). The obtained result was (0.23), which indicates a weak correlation.

Finally, Table 8 describes the correlation between the two rankings based on classifying the respondents who have the highest years of experience in software development on the same software project. The results show that the correlation ranged from very weak to weak with preference when the respondents have more expertise in software development in general.

## 6. Threats to Validity

As for threats to **construct validity** the set of new God Classes detected in the new classes in the target versions of the software projects, because the target versions include new classes that were never found in the previous versions. Therefore, these God Classes will never be detected. To this end, further work has to be done by including more intermediate versions that might resolve the problem. The selection of only one previous version of the project is an essential threat because it might affect the strategy to compute the appropriate parameters of each criterion. Another threat is the unbalanced number of parameters used to represent the criteria, precisely, the developer context. We plan to replicate the study by including more parameters to this criterion. To take all the factors expressed in this list, a strategy should consider some infor-

**Table 7**
Correlation values between our prioritization technique and different categories of respondents: Administrator (Admin), Architect (Arch), Developer (Dev), Tester (Test), Regular Contributor (RegCont.) and Sporadic Contrib (SprCont.) who agree with the technique according to their role in the project.

| Respondents Cat. | Admin. | Arch. | Dev. | Test. | RegCont. | SprCont. |
|---|---|---|---|---|---|---|
| # Respondent | 1 | 2 | 8 | 1 | 2 | 3 |
| Correlation | NA | −0.07 | 0.08 | 0.8 | 0.71 | −0.06 |

**Table 8**
Correlation values between our prioritization technique and the different categories of respondents who have the highest experience in software development and in the same software project regarding the ranking positions.

| Experience | Exp. Software Development | In the same project |
|---|---|---|
| # Respondent | 8 | 10 |
| Correlation | 0.25 | 0.15 |

mation from issue trackers. This aspect could be essential to consider in future work.

The project contributors are the main threat to **internal validity**. Each project contributor has a set of different achievements based on their background and experience to play a particular role in the project team, such as admin, tester, designer, architect, and core developer. In other cases, the contributors might be sporadic and do not have a deep knowledge of the project details. Another threat is related to the number of core software developers responding to the survey and those with a high experience in the selected project. Different results may be obtained when this number increases.

The main concern of **external validity** is the generalizability of results. In this study, we analyzed the java projects from different domains using the five detection tools developed based on a metrics-based approach. In addition, only one type of Design Smell (God Class) has been selected to conduct the study. Another threat is that the developers only ranked the first five God Classes because asking the developers to rank the whole list of God Classes is time-consuming and costly. Moreover, the number of developers in the study depends on the software team and contributors, which varies from one project to another and might not be enough in some projects.

Finally, the main threat to **conclusion validity** is related to the absence of quantitative analysis using the well-known statistical techniques because of the number of developers who evaluated the God Classes of each project. Moreover, the number of God Classes assessed and ranked by the developers is another threat. We think the replication of the study with many developers and analyzing the results quantitatively are significant.

## 7. Conclusions and Future Work

We proposed a Design Smell Prioritization technique. Specifically, for God Class Design Smell, based on a combination and merging of three criteria (Historical Information, Design Smell Density, Developer Context), we assumed essential and the developers should be taken into account. Then, the technique was carried out in the context of two different versions of 24 open source software systems and empirically evaluated by professional developers from the same software system teams. Besides, the developers were asked to determine the factors they considered for ranking the God Class Design Smell. Spearman's correlation coefficient was used to examine the ranking results. The results show a weak correlation between the rankings proposed by our technique and the ranking obtained by evaluators in general. Better results were obtained with the evaluators who have more years of experience. Also, the Task Relevance and Module Importance were the most critical factors that the evaluators took into account during the God Class prioritization.

It would be interesting to modify the technique to consider different types of Design Smells and include parameters that analyze the relationship between them. It would also be interesting to analyse several versions of the same software, instead of just one, to improve the evaluation of the evolution of the software by expanding the measures used in the historical information of changes.

## Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

Alkharabsheh, K., 2021. An empirical study on the co-occurrence of design smells in the same software module:god class case study, in. In: 2021 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), pp. 1–6. https://doi.org/10.1109/JEEIT53412.2021.9634144.
Alkharabsheh, K., Almobydeen, S., Taboada, J.A., Crespo, Y., 2016. Influence of nominal project knowledge in the detection of design smells: An exploratory study with god class. Int. J. Adv. Stud. Computers, Sci. Eng. 5, 120.
Alkharabsheh, K., Crespo, Y., Manso, E., Taboada, J.A., 2018. Software design smell detection: a systematic mapping study. Software Qual. J., 1–80
Alkharabsheh, K., Alawadi, S., Crespo, Y., Manso, M.E., Taboada, J.A., 2021. Analysing agreement among different evaluators in god class and feature envy detection. IEEE Access 9, 145191–145211.
Alkharabsheh, K., Crespo, Y., Delgado, M.F., Viqueira, J.R.R., Taboada, J.A., 2021. Exploratory study of the impact of project domain and size category on the detection of the god class design smell. Softw. Qual. J. 29, 197–237.
Alkharabsheh, K., Alawadi, S., Kebande, V.R., Crespo, Y., Delgado, M.F., Taboada, J.A., 2022. A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of god class. Inf. Softw. Technol. 143, 106736.
Arcoverde, R., Garcia, A., Figueiredo, E., 2011. Understanding the longevity of code smells: preliminary results of an explanatory survey, in. In: Proceedings of the 4th Workshop on Refactoring Tools. ACM, pp. 33–36.
Arcoverde, R., Guimarães, E., Macía, I., Garcia, A., Cai, Y., 2013. Prioritization of code anomalies based on architecture sensitiveness. In: Software Engineering (SBES), 2013 27th Brazilian Symposium on. IEEE, pp. 69–78.
W.J. Brown, R. Malveau, H.W.M. III, T.J. Mowbray, Antipatterns refactoring software, architectures and projects in crisis, John Wiley & Sons Inc, 1998.
L.P. da S Carvalho, R. Novais, M. Mendonça, Investigating the relationship between code smell agglomerations and architectural concerns: Similarities and dissimilarities from distributed, service-oriented, and mobile systems, in: Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse, ACM, 2018, pp. 3–12.
Curtis, B., Sheppard, S.B., Milliman, P., 1979. Third time charm: Stronger prediction of programmer performance by software complexity metrics. In: Proceedings of the 4th international conference on Software engineering. IEEE Press, pp. 356–360.
da Silva Sousa, L., 2016. Spotting design problems with smell agglomerations. In: Proceedings of the 38th International Conference on Software Engineering Companion. ACM, pp. 863–866.
Fontana, F.A., Ferme, V., Zanoni, M., Roveda, R., 2015. Towards a prioritization of code debt: A code smell intensity index. In: 2005 IEEE 7th International Workshop on Managing Technical Debt (MTD). IEEE, pp. 16–24.
Fontana, F.A., Ferme, V., Zanoni, M., 2015. Filtering code smells detection results. In: Proceedings of the 37th International Conference on Software Engineering-Volume 2. IEEE Press, pp. 803–804.
F. Palomba, G. Bavota, M.D. Penta, R. Oliveto, A.D. Lucia, Do they really smell bad? A study on developers' perception of bad code smells, in: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014, 2014, pp. 101–110.

Islam, M.R., Al Maruf, A., Cerny, T., 2022. Code smell prioritization with business process mining and static code analysis: A case study. Electronics 11, 1880.

Jay, G., Hale, J.E., Smith, R.K., Hale, D.P., Kraft, N.A., Ward, C., 2009. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. JSEA 2, 137–143.

Jbara, A., Matan, A., Feitelson, D.G., 2014. High-mcc functions in the linux kernel. Empirical Software Eng. 19, 1261–1298.

K. Alkharabsheh, Y. Crespo, E. Manso, J. Taboada, Sobre el grado de acuerdo entre evaluadores en la detección de design smells, in: Jornadas de Ingeniería del Software y Bases de Datos, 2016, pp. 143–157.

K. Alkharabsheh, Y. Crespo, E. Manso, J. Taboada, Comparación de herramientas de detección de design smells, in: Jornadas de Ingeniería del Software y Bases de Datos, 2016a, pp. 159–172.

K. Alkharabsheh, Y. Crespo, M.E. Manso, J.A. Taboada, Factors that developers take into account when prioritizing smells for their correction: conclusions after a reply., in: JISBD2019, SISTEDES, 2019, pp. 1–6. URL:http://hdl.handle.net/11705/JISBD/2019/ 089.

Kaur, A., Jain, S., Goel, S., Dhiman, G., 2021. Prioritization of code smells in object-oriented software: A review. Materials Today: Proceedings.

K. et al., Survey questionnaire, 2022. URL:https://gitlab.inf.uva.es/yania/prioritization-factors/raw/1a4bc19d51625682eae019c9b3c500c389b03e13/PrioritizationSurvey.pdf?inline=false.

Landman, D., Serebrenik, A., Vinju, J., 2014. Empirical analysis of the relationship between cc and sloc in a large corpus of java methods. In: Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on. IEEE, pp. 221–230.

R. Marinescu, Assessing technical debt by identifying design flaws in software systems, IBM Journal of Research and Development 56 (2012) 9–1.

Oizumi, W.N., Garcia, A.F., Colanzi, T.E., Ferreira, M., Staa, A.V., 2015. On the relationship of code-anomaly agglomerations and architectural problems. J. Software Eng. Res. Dev. 3, 11.

Oizumi, W., Garcia, A., da Silva Sousa, L., Cafeo, B., Zhao, Y., 2016. Code anomalies flock together: Exploring code anomaly agglomerations for locating design problems. In: Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on. IEEE, pp. 440–451.

Oizumi, W., Sousa, L., Garcia, A., Oliveira, R., Oliveira, A., Agbachi, O.I.A.B., Lucena, C., 2017. Revealing design problems in stinky code: A mixed-method study. In: Proceedings of the 11th Brazilian Symposium on Software Components, Architectures, and Reuse, SBCARS '17. ACM, New York, NY, USA, pp. 5:1–5:10. https://doi.org/10.1145/3132498.3132514. URL:http://doi.acm.org/10.1145/3132498.3132514.

Ouni, A., Kessentini, M., Bechikh, S., Sahraoui, H., 2015. Prioritizing code-smells correction tasks using chemical reaction optimization. Software Qual. J. 23, 323–361.

Peters, R., Zaidman, A., 2012. Evaluating the lifespan of code smells using software repository mining. In: Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on. IEEE, pp. 411–416.

Pirie, W., 2004. Spearman rank correlation coefficient. Encyclopedia Stati. Sci. 12.

Sae-Lim, N., Hayashi, S., Saeki, M., 2016. Context-based code smells prioritization for prefactoring. In: Program Comprehension (ICPC), 2016 IEEE 24th International Conference on. IEEE, pp. 1–10.

N. Sae-Lim, S. Hayashi, M. Saeki, Revisiting context-based code smells prioritization: on supporting referred context, in: Proceedings of the XP2017 Scientific Workshops, ACM, 2017, p. 3.

Sae-Lim, N., Hayashi, S., Saeki, M., 2017. How do developers select and prioritize code smells? a preliminary study. In: Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on. IEEE, pp. 484–488.

Saranya, C., Manikandan, G., 2013. A study on normalization techniques for privacy preserving data mining. Int. J. Eng. Technol. (IJET) 5, 2701–2704.

Singh, R., Bindal, A., Kumar, A., 2021. Software engineering paradigm for real-time accurate decision making for code smell prioritization. In: Data Science and Innovations for Intelligent Systems. CRC Press, pp. 67–93.

S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A.D. Pace, C. Marcos, Identifying architectural problems through prioritization of code smells, in: Software Components, Architectures and Reuse (SBCARS), 2016 X Brazilian Symposium on, IEEE, 2016a, pp. 41–50.

Y. Tashtoush, M. Al-Maolegi, B. Arkok, The correlation among software complexity metrics with case study, arXiv preprint arXiv:1408.4523 (2014).

Verma, R., Kumar, K., Verma, H.K., 2021. A study of relevant parameters influencing code smell prioritization in object-oriented software systems. In: 2021 6th International Conference on Signal Processing, Computing and Control (ISPCC). IEEE, pp. 150–154.

Vidal, S.A., Marcos, C., Díaz-Pace, J.A., 2016. An approach to prioritize code smells for refactoring. Automated Software Eng. 23, 501–532.

Yamashita, A.F., Moonen, L., 2012. Do code smells reflect important maintainability aspects? In: 28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23–28, 2012, pp. 306–315.