Contents lists available at ScienceDirect

Information and Software Technology

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

Selection of human evaluators for design smell detection using dragonfly optimization algorithm: An empirical study

Sultan M. Al Khatib [a], Khalid Alkharabsheh [a], Sadi Alawadi [b,c,*]

[a] Department of Software Engineering, Prince Abdullah bin Ghazi Faculty of Information and Communication Technology, Al-Balqa Applied University (BAU), Al-Salt, 19117, 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

ARTICLE INFO

Keywords:
Software quality
Design smell detection
Dragonfly Algorithm
Search-based software engineering
Optimization
God class
Empirical study

ABSTRACT

Context: Design smell detection is considered an efficient activity that decreases maintainability expenses and improves software quality. Human context plays an essential role in this domain.
Objective: In this paper, we propose a search-based approach to optimize the selection of human evaluators for design smell detection.
Method: For this purpose, Dragonfly Algorithm (DA) is employed to identify the optimal or near-optimal human evaluator's profiles. An online survey is designed and asks the evaluators to evaluate a sample of classes for the presence of god class design smell. The Kappa-Fleiss test has been used to validate the proposed approach.
Results: The results show that the dragonfly optimization algorithm can be utilized effectively to decrease the efforts (time, cost) of design smell detection concerning the identification of the number and the optimal or near-optimal profile of human experts required for the evaluation process.
Conclusions: A Search-based approach can be effectively used for improving a god-class design smell detection. Consequently, this leads to minimizing the maintenance cost.

1. Introduction

In recent decades, software systems have grown widely in size, functionality, and complexity. Consequently, maintaining software quality is one of the essential issues that has taken the attention of software engineers and industries. Maintaining quality requires continuous actions to identify and detect the weak parts in design and implementation. In the literature, these parts are called Design Smell [1]. The design smell does not lead to compilation or runtime errors [1] but negatively affects the software quality [2,3].

Design smell detection is a fundamental activity that assists in improving software quality [3–6]. Several approaches, methods, and techniques have been suggested for design smells detection that ranged from manual inspection [7–12] to fully-automated, such as metric-based [13–18], rule-based [19–21], and machine learning [22–26]. Number of approaches have been developed into detection tools, such as iPlasma [15], DECOR [17], and JDeodorant [27]. The existing tools can analyze software projects developed in various languages and detects a wide range of design smells in the source code or design.

Despite the accuracy of proposed approaches and detection tools, there exists a low degree of agreement on their detection results [28–32]. The different lists of detected design smells are considered a massive challenge for software developers concerning maintainability due to the variety and negative influence on the software systems [3, 33–35]. Therefore, it should be analyzed to determine the true positive design smells that threaten the software quality. To improve the level of agreement on design smell detection between automatic evaluators, several studies addressed the role of human subjectivity and how it can contribute in this context [7–12]. The result of studies has shown an agreement on design smell detection between a group of human evaluators, who have specific profiles concerning experience, working background, and role in the software projects team. They conclude that human subjectivity is a substantial factor.

The work presented in this paper considers human context, confirming what has been concluded by previous studies on this subject, as a significant aspect for practitioners of software engineering to have in the software industry, which leads to reducing future technical debts and maintenance costs. Therefore, the opinions of human evaluators

* Corresponding author at: Department of Information Technology, Uppsala University, 75105, Uppsala, Sweden.
E-mail addresses: s.al-khatib@bau.edu.jo (S.M. Al Khatib), khalidkh@bau.edu.jo (K. Alkharabsheh), sadi.alawadi@it.uu.se, sadi.alawadi@hh.se (S. Alawadi).

https://doi.org/10.1016/j.infsof.2022.107120
Received 30 June 2022; Received in revised form 12 November 2022; Accepted 22 November 2022
Available online 24 November 2022

should be considered during the development of design smell detection techniques and the software development cycle in general. In this context, authors of [36–40] have shown that they can use search-based techniques effectively to improve the activity of design smells detection in software systems. For this purpose, we focus on optimizing the selection of human evaluators for design smell detection using search-based approaches. The proposed approach utilizes the Dragonfly (DA) heuristic optimization technique to determine the optimal or near-optimal human evaluator's profile. For this purpose, we have designed an online questionnaire that asks human evaluators to validate the presence of god class design smell in a set of classes of an open-source project. The sample has been previously detected as a god class by a group of detection tools [32,41]. The Kappa-Fleiss test is used to evaluate the results.

The main contribution of this work is summarized as follows:

- Exploiting of human evaluators profile-based identification using Dragonfly algorithm.
- Formalizing the selection of human evaluators for design-smell detection as an optimization problem.
- An empirical assessment for the degree of agreement between the selected human evaluations by the proposed approach and the survey outcomes.
- Providing a replication package that involves all necessary information for study replication [42].

As for the remainder of this paper, Section 2 presents related works focused on search-based approaches and the human context concerning design smell detection. Next, Section 3 discusses the problem statement concerning design smell, target system, problem formulation, optimization algorithm, and the designed survey. Then, Section 4 describes the proposed methodology. Afterwards, Section 5 analyzes and discusses the results. Finally, Section 7 presents our conclusions and future work.

2. Related work

In this section, we addressed related works focused on search-based methods for design-smell detection and the role of human context in smell detection activity.

The author of [36] used genetic programming to identify refactoring opportunities related to design antipatterns through the example approach. Their study analyzed a sample of design smell examples extracted from well-known systems and utilized this information to extract detection rules. Also, in [37], a heuristic approach using a cooperative Parallel Evolutionary Algorithm (P-EA) is used to detect eight web service antipatterns. The technique used genetic programming to obtain the detection rules from a set of examples of web service antipatterns using coupling and other types of metrics. The experiment was executed using eight antipatterns (god object, Fine-grained, Chatty, Data, Ambiguous, Redundant PortTypes, CRUDy Interface, and Maybe It is Not RPC). The results showed that the proposed approach effective in web service antipattern detection with high precision and recall values. In [38], a novel approach to detect design smells using evolutionary data mining has been proposed. Heuristics algorithms are used with the change history of software to extract the association rules to identify design smells. The study used five projects and three design smells (Duplicated Code, Divergent Change, and Shotgun Surgery). The approach was compared with DECOR, JDeodorant, and SonarQube. The finding showed that the approach achieved high precision and recall. Next, [39] employed a Euclidean distance-based Genetic Algorithm and Particle Swarm Optimization (EGAPSO) for design smell detection. The idea of their work is based on determining the proper threshold value and defining the detection rule using EGAPSO. The proposed approach was validated using two software systems and five design smells (The Blob, Data class, Feature Envy, Functional Decomposition, and Spaghetti Code). After comparing the EGAPSO with other approaches, such as GA, DECOR, and MOGP, the outcome showed it is effective. Finally, in [40], a novel search-based approach has been proposed to detect design smell. The authors used the Whale Optimization Algorithm (WOA) to find the optimal detection rules for design smells in medium and large sizes software. The experiments are carried out on five software systems to detect nine types of smells (Feature Envy, Data class, Lazy class, Long Parameter List, The Blob, Long Method, Spaghetti Code, Functional Decomposition, and Parallel Inheritance). The results demonstrate high precision and recall values.

On the other hand, several works addressed the role of human context in smell detection activity. In [7], human evaluators were asked in two different experiments about the existence of three design smells (Feature Envy, Long Parameter List, Long Method) and if those smells should be corrected. The number of participants in both experiments was 46 and 36, respectively, in which most of the evaluators were master's degree students. The results have shown a high agreement between evaluators in the case of Long Method and Long Parameter List detection. In contrast, a low agreement about Feature Envy and if the design smell will be refactored. Also, [8] experimented with asking evaluators about the existence of 23 design smells in various artifacts developed in a small software development company. Three developers evaluated each module on average. The number of developers who participated in the experiment was 18. The results show that the leader developers detected structural design smells while the regular developers detected duplicate and dead code smells. The authors conclude that subjectivity played a role in the evaluation process. Next, [9] conducted a study involving six developers in maintaining two software systems, taking into account the opinion of two external evaluators before and through the maintainability process. The results have shown an agreement between developers and evaluators regarding substantial factors affecting software maintainability. Furthermore, they found a partial correlation between these factors and the detection of some types of design smell. In [10], a survey was designed to explore the significance of design smells from the point of view of software developers. If they do not care, do the reasons concerned with the unawareness of developers, the irrelevance of the design smell concepts, or the lack of appropriate detection tools? To this end, 85 professional developers participated in the survey. The results show that 32% of developers had never heard of design smell or similar terms. 22% answered that they heard or read about them, but they did not know what it meant. 21% see the concept but do not apply it. Only 18% understand the topic and apply it in their work. Moreover, only two developers have used the detection tools. The authors concluded the need for training, and awareness, in addition to the appropriate tools with desirable features. Another study carried out by [11] involved a group of 34 participants, including master's students (15), open-source developers (10), and developers who work in software industries (9). The study was performed to determine to what extent developers look to the design smells as design problems that should be solved and which are the most harmful. They concluded that there is a misuse or misunderstanding of object-oriented design principles. In [12,43], the authors performed a survey to identify the agreement between human evaluators on the presence of two types of design smells. A group of 92 persons of different profiles participated in the survey. The results have shown an evaluator profile that leads to better detection agreements. Furthermore, the profile indicates a better matching between the experienced developers in the size and complexity concepts while a better coincidence on the principles of object-oriented design between little experience developers. Therefore, they concluded more need for training on design smells. In addition, it is necessary to consider the opinion of human evaluators in reviewing the code written by others and confirm the results of detection tools. Next, in [44], they addressed how the developers discuss and perceive the design smells (code smells and antipatterns), the correction procedures to deal with smells, and the technical limitation faced by developers when working with smells. Finally, the authors conducted a large-scale study in three stack exchange sites in which more than

4000 posts were analyzed quantitatively and qualitatively. The findings show that most developers focused on hot groups of smells, such as god class, data class, duplicated code, and spaghetti code detected in Java, C#, and JavaScript. Moreover, there is a gap between the researchers and developers concerning the design smell detection, which makes the questions open in this context. Also, the effectiveness of design smell detection collaboratively versus individual is studied in [45]. To this end, a controlled experiment that included 34 developers and five team leaders of software projects has been conducted based on a specific scenario. The chosen developers with different experiences, beginners and professionals. The results show that collaborative smell detection has more precision compared with individuals, and this approach is strongly recommended by software project leaders to detect design smells. After that, in [46], analyzed the code smell co-occurrence removal influence on internal quality factors from the human context perspective. A study by 14 developers was conducted over three months on five closed-source software systems to analyze the refactoring operations during the removal of 60 cases of code smell co-occurrences. From the developer's perspective, the findings show that correction of the code smell pair (Dispersed Coupling–God Class) enhancement some internal quality factors. Also, the refactoring operations on code smell co-occurrences are considered a challenge because of the problem of understanding the source code and the complexity of refactoring operations. Moreover, the developer's worried about identifying and refactoring the true positive code smells. In parallel, the authors of [47] employed the code reviews technique to detect code smell empirically. A set of keywords have been identified to mine code smells from the discussions of the source code review of 2 OpenStack systems. The results indicate code review method is not standard in code smell detection, and the reviewers assist developers by providing some recommendations to apply the refactoring. They conclude that developers should follow good practices in object-oriented programming to reduce the occurrence of code smells. Also, the code review method is trustworthy to developers.

Throughout the literature, a conclusion can be made on how subjectivity factors of human context in design smell detection play a vital role in detecting various types of code smells. The extensive discussions made by (references number [7–12,43–47]) provide that subjectivity factors have different implications on how the evaluators perform and to what smell they can detect in the evaluation process. For instance, [8,12,43] indicate to what extent can the experience, especially on Object-Oriented (OO), provides significant guidance on programming practices. Those studies show that evaluators with high experience were able to detect smell according to the size and complexity of classes, whereas those with low experience have focused on OO standards and good practices, i.e. code duplicate and long methods. The need is, accordingly, for a selection mechanism of competent evaluators to perform design smell detection. This is a hard problem, given the large number of human subjectivity features to consider, which requires incorporating human context to optimize the evaluator's selection. From what can be seen in the literature is the lack of proposing the selection of design smell evaluators as an optimization problem. Most of the previous works focused on using search-based techniques to generate design smell detection rules without taking into account the essential role of human context, as well as refactoring systems. In this sense, we need to discover whether employing a search-based optimization algorithm is beneficial for reducing the number of evaluators involved in the evaluation process for the sake of reducing the evaluation time and cost. The proposed approach, therefore, combines search-based techniques with human context for design smell detection.

3. Problem statement

3.1. Design smell selection

According to the systematic mapping study of [3], more than 600 design smells have been detected either in the source code or the design of software systems. From the software engineering community's point of view, these smells vary in their negative influence and degree of importance. The scope of this study focused on the god class design smell, which is one of the most design smell types that have taken the attention of the community [3,12,41,43,48–50] due to the negative characteristics that affect different quality factors, particularly maintainability, stability, and complexity [2,3]. God class is a class that has more functionality and responsibilities, involves many methods, and has a large size (thousands of number lines of code). Therefore, it tends to be more complex. In some cases, god class maintenance activity can become difficult. The reasons return to the nature of the god class definition, where various responsibilities are centralized in one software module that can be intersected in their source code [49]. In addition, the maintenance process might be costly and needs more effort that exceeds the assigned budget. Mainly when the maintenance is time-consuming due to the size of the task and their implementation includes error-prone [51,52]. Other concepts are used in the literature to describe the god class, such as Large class and The Blob according to Fowler [53], and Brown definition [2] respectively. Most of the existing detection tools have focused on god class detection along with other types of smells due to the nature of smell that has taken the attention of developers more than other types. Different techniques and strategies have been used for smell detection in these tools, such as clustering algorithms, rule-based, and metric-based strategies. In this experiment, we selected a set of five detection tools which are the most cited and used in this context according to [3] and include DECOR, Together, iPlasma, JDeodorant, and PMD. For example, iPlasma and PMD use the exact mechanism defined by [53], but the threshold values differ. In addition, the defined strategy uses three metrics combined in one rule to detect god class. The defined metrics include Weighted Methods per Class (WMC), Access to Foreign Data (ATFD), and Tight Class Cohesion (TCC). The detection rule is as follows:

```
WMC<= WMC_VERY_HIGH && ATFD > FEW_ATFD && TCC < TCC_VERY_LOW
```

In PMD, the threshold values were (WMC <= 47, ATFD > 5, TCC < 1/3) while in iPlasma were (WMC <= 20, ATFD > 4, TCC < 1/3). Borland Together uses a group of metrics from [54], and [55] where the threshold values are not published due to the tool being commercial. DECOR is a collection of rules (metrics, semantic, structural and association). The metrics obtained from [56] were Number of Attributes Declared (NAD), Number of Methods Declared (NMD), and Lack of Cohesion of Methods (LCOM5). At the same time, JDeodorant uses the agglomerative clustering technique to obtain clusters of features (properties and methods) for each class to detect the "extract class" refactoring possibilities.

3.2. Target system

To conduct the study, we selected the GanttProject version 2.0.10, a well-known software system in the design smell detection context. GanttProject is application software used for scheduling and managing projects. It is an open-source system implemented in Java, belongs to the medium–large category, has a long maintenance history, and is licensed as free software. The source code of GanttProject is available on the web in different repositories, such as GitHub and SourceForge, and can be freely downloaded.[1] Table 1 shows the full details of the project.

3.3. Problem formulation

Our problem can be represented as searching for the least number of evaluators competent to perform the evaluation job in a short time from a set of available ones. The set of available evaluators is

[1] http://ganttproject.biz.

Table 1
Characteristics of GanttProject version 2.0.10.

# Package	# class	# Method	# Line of code
52	621	5047	66,540

represented as $E = \{e_1, e_2, \ldots, e_n\}$ of size n. In addition, the competency characteristics we consider while searching for the best fit evaluator(s) incorporate five characteristics from the evaluator's profile. These characteristics are the role (s)he plays during software development, to which software domain his/her expertise is on, programming languages (s)he is an evaluator with, Object-Oriented Programming experience, and code smell detection experience.

There are different studies that each has shown the importance of one or some of these characteristics to be considered as in [8,10–12,43]. We can think of it as in software testing and verification; a third party is sometimes required to cover up aspects that developers could not do, so evaluators have to have the knowledge and expertise on the job they are about to do that is acquired by their working time. Here, we consider characteristics of which they play a certain perspective according to the working experience. For instance, an evaluator would give much attention to some qualities that could affect the software according to his/her development role perspective. Same as a designer who would give much attention to details where aspects of user interfaces may be affected. Therefore, each e in E provides a set of values associated with these five competency characteristics.

However, these characteristics are divided into two subsets based on how they will be used in the search process. In our problem, one part of the search process requires matching between the competencies needed for the software under investigation represented by S and what E has to offer. The second part incorporates measuring the evaluator's productivity and finding those capable of completing the evaluation job in a short time.

The first subset, represented as set C^1, includes the characteristics we need to match with the details of S. Here, both software domain D and programming language(s) P are considered, where S is represented as $S = \{S_D, S_P\}$ and C^1 as $C^1 = \{e_D, e_P\}$. Rather using a constrained optimization, we relax the selection criterion and provide a penalty pen that shows the fitness for an e to be selected. pen is designed to cumulatively hold each mismatch for each selected e for each domain and programming language characteristics between S and C^1. Therefore, the cumulative value of pen will contribute on the final estimate of the evaluation time span. If e holds the competencies required for S, then no penalty will be associated. Otherwise, a penalty will be included for those whose characteristics do not match. Accordingly, the matching process between e and S is formed by the following steps.

First, we compare whether the evaluator would possess the competency of domain expertise according to S. This is depicted by the following Eq. (1):

$$pen_D(e_y) = \begin{cases} 0, & If\ e_y(C_1^1) = S_1 \\ 1, & If\ e_y(C_1^1) \neq S_1 \end{cases} \tag{1}$$

In the above Eq. (1), we can see that a value of 0, representing no penalty, will be associated with evaluator y if the domain characteristic in $e_y(C_1^1)$ matches the one in S_1. In case of no match for this characteristic between S_1 and C_1^1, a value of 1 will be added to this evaluator as a penalty. Moreover, programming languages expertise of the evaluator will be compared with what S is developed by. The following Eq. (2) represents this part of the process.

$$pen_P(e_y) = \begin{cases} 0, & If\ e_y(C_2^1) = S_2 \\ 1, & If\ e_y(C_2^1) \neq S_2 \end{cases} \tag{2}$$

By Eq. (2), a penalty will be associated to evaluator y if the programming languages characteristic in $e_y(C_2^1)$ does not match the one in S_2. Noteworthy that an evaluator may have more than one programming language in his/her experiences. Consequently, C_2^1 may fold a subset representing these languages. Consequently, the overall penalties identified for evaluator e_y for these two types can be represented as the following Eq. (3).

$$pen(e_y) = pen_D(e_y) + pen_P(e_y) \tag{3}$$

It can be seen by Eq. (3) that the pen of evaluator e_y may be summed up with a value of 0 as a highly fitted one, 1 for penalizing for one characteristic mismatch, or 2 for no matching for both characteristics.

In addition to the matching criterion, we need to measure how well the selected evaluators will perform their evaluation job in terms of productivity. The second subset, represented by C^2, incorporates the characteristics from which we believe the level of improvement in the evaluator productivity can be identified. These characteristics are the role, OOP and code smell detection experiences. The representation of OOP and code smell detection experiences can be either by the value of 1 for a senior, 2 for a junior, or 3 for a fresh. The role, on the other hand, can be represented by the value of 1 as (professor/lecturer/instructor), 2 for developer, 3 for tester, 4 for system architect, 5 for designer/software engineer, 6 for system administrator, and 7 for related CS student. Therefore, the lower the value of these representations of role, OOP, and code smell detection experiences, the higher the experience is.

Two assumptions are forming the identification of productivity improvement. First, the more experience that e has on each productivity characteristic, the more progress his/her productivity will be. Moreover, we assume that when the representation of characteristics in C^2 provides a high level of experience, then the productivity of e to that characteristic is equal to 1, which means that this evaluator has a 100 percent productivity on that characteristic. By these two assumptions, we need to reach the best subset of E, whose productivity is improved by their experiences.

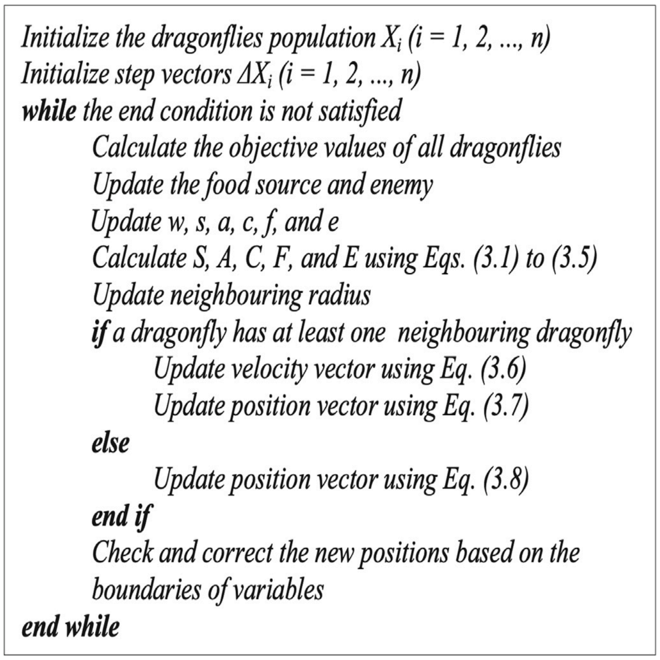
On the contrary, productivity can also be reduced or weakened when the value associated with the characteristic is low. For instance, when an e has the value of 3 for code smell detection, then his/her productivity for this characteristic should be reduced by a percentage to represent a fresh evaluator. For this reason, given the limited range of values representing the evaluator's characteristics, such as from 1 to 3 or from 1 to 7, the improvement can be formed by a logarithmic function. This function will consider the amount of reduction that should be made on productivity by the level of experience on each characteristic using the log of base 10. The improvement of productivity imp for each e can, accordingly, be measured by the following Eq. (4):

$$imp(e_y) = \sum_{a=1}^{3} (1 - log\ \ e_y(C_a^2)) \tag{4}$$

It can be seen by Eq. (4) above that imp of e_y is the cumulative sum of productivity values for the three characteristics. The productivity improvement is the resulting value of subtracting the percentage of reduction based on the values associated with each characteristic from 1. In this representation, a fully experienced e can be three times more productive than others when his/her level of experience in the three characteristics is high.

3.4. Dragonfly optimization algorithm

According to the social interaction behaviors of dragonflies in nature, Mirjalili et al. developed the (DA) Algorithm in 2016 [57]. Therefore, the algorithm is a nature-inspired technique with three main processes to follow. One of these processes mimics the navigation of dragonflies between different locations. The second process goes further into the way of searching for food. Finally, the third process imitates the dragonflies' behavior when it needs to bypass enemies. Moreover, they consider in their proposal the movement of dragonflies in a swarm to be, for their similarity, the exploration and exploitation



```
Initialize the dragonflies population $X_i$ ($i = 1, 2, ..., n$)
Initialize step vectors $\Delta X_i$ ($i = 1, 2, ..., n$)
while the end condition is not satisfied
    Calculate the objective values of all dragonflies
    Update the food source and enemy
    Update w, s, a, c, f, and e
    Calculate S, A, C, F, and E using Eqs. (3.1) to (3.5)
    Update neighbouring radius
    if a dragonfly has at least one neighbouring dragonfly
        Update velocity vector using Eq. (3.6)
        Update position vector using Eq. (3.7)
    else
        Update position vector using Eq. (3.8)
    end if
    Check and correct the new positions based on the
    boundaries of variables
end while
```

Fig. 1. Pseudo code of the DA algorithm [57].

of meta-heuristics as the main phases for optimization. Thus it is classified as a swarm intelligence (SI) based technique. These movements, however, are recognized by [57] as static behavior (exploration) for sub-swarms movement and dynamic behavior (exploitation) for larger swarm movement. These two phases allow for expanding the search in the solution space. Furthermore, this algorithm models three primitive swarm principles: individual separation, alignment, and cohesion. These behaviors are intended to be factors that aid in the updating of individuals' positions in swarms.

Fig. 1 depicts the algorithm's search process. As seen in the figure, the algorithm will generate initial random solutions representing the dragonfly population. One of the algorithm's parameters, named search agents, manages the population size. The second step involves two additional parameters of the algorithm to represent the dragonflies' movement and direction: the lower and upper boundaries. The algorithm will initialize random step vectors in the second step by these two parameters. However, each step vector's calculation depends on five swarming factors to balance the exploration and exploitation. These factors are separation, alignment, cohesion, food position, and an enemy position.

In each iteration, the position updating process is repeated. It involves updating the values of swarming factors and measuring the fitness of all dragonflies according to the objective function. This process is repeated until the end criterion is satisfied. The maximum number of iteration parameters determines the end criterion. Moreover, the dimensions parameter, which represents the number of features in the data, specifies the number of dragonflies. The algorithm incorporates a flight mechanism for a random walk named levy to improve stochastic positioning. This flight mechanism allows for more search space exploration when there are no neighboring solutions. The reader can find more information at [57].

Solution Representation:

The solution structure to this problem is represented by a vector named E^*, which has the same size as E. The representation for each selected element e^* in E^* is binary, having the value of 1 for those chosen and 0 otherwise. For n evaluators, the solution representation is demonstrated by the Fig. 2.

$$\begin{array}{cccccc} & e_1^* & e_2^* & e_3^* & \dots & e_n^* \\ E^* = & \boxed{1} & \boxed{0} & \boxed{1} & \boxed{\dots} & \boxed{0} \end{array}$$

Fig. 2. Solution representation.

In Fig. 2, it can be seen that only evaluator e_1^* and e_3^* are selected. These evaluators are selected according to the search-based algorithm stochastic processes described in Section 3.4. The size of the formed team by E^* is, therefore, dynamic. Accordingly, the resulting evaluator team will stochastically be formed by this vector and then need to be assessed by the fitness function. The fitness function will help, by its outcome value, in deciding whether to accept the solution according to the selected evaluators or not.

Fitness Function:

Our fitness function is a cost function, and it incorporates two steps. Now, for each selected e^* in E^*, forming the evaluation team, the first step is to measure their suitability to perform the evaluation job according to the description of S. This can be done by matching the competencies for every selected e^* in E^* with the description of S using Eqs. (1), (2), and (3). By these equations, we relax the constraints and provide a penalty in case of a mismatch. Consequently, we will either have the value of $pen \geq 1$ by Eq. (3) to penalize the outcome of the fitness function for each mismatch or 0 for a fitted team.

Moreover, we need to estimate how productive the formed team by E^* is to finish the evaluation job as early as possible. Productivity characteristics for each selected e^* in E^* are, therefore, used by the second step, which incorporates Eq. (4). By Eq. (4), the improvement of productivity for all selected e^* in E^* can be measured, and in accordance with the size of S, the estimate of the evaluation time span eT can be provided. However, as those two aspects of productivity and size will work well when we have a fully fitted team, we might have, on the other hand, a resulted team by the optimizer that consists of one or more mismatched e^* to S. Consequently, the value of pen will also be considered to penalize eT for any mismatch. This mismatch is illustrated when $pen \geq 1$, which should add an extra time to eT estimate. This eT estimate is depicted by the following Eq. (5):

$$eT = \frac{size_s * (1 + \sum_{o=1}^{n} pen(e_o) * e_o^*)}{\sum_{o=1}^{n} imp(e_o) * e_o^*} \tag{5}$$

It can be seen in Eq. (5) above that the evaluation time span eT is calculated as the product of the size of S, in terms of lines of code, by the overall penalty pen of the selected evaluators in E^*. The resulting value is divided by the cumulative production of the selected evaluators in E^*. This type of time estimate is similar to the ones used for resource allocation with consideration to the resource attributes, which are used in many incarnations and discussed by [58–60]. When the team formed by E^* is fully fitted to S, then the cumulative pen value will be equal to 0 and eT can be as the size of S over productivity. Otherwise, we will end up, for example, by a twice or thrice value of eT. Across the different stochastically created permutations by the optimizer, we depend now on eT to find the minimal estimated eT that represents the optimal or near-optimal solution.

3.5. Survey questionnaire

As mentioned above, DA is a population algorithm that searches in a population until the optimal solution is found. For this purpose, to construct the population (in our case, the human evaluators), we designed a web-based survey that asks the evaluators to detect the god class design smell in a sample of classes of the target project GanttProject. The selected sample has been detected as god class using the chosen group of detection tools. The designed survey consists of two parts. The first part includes a group of questions that make up the evaluator profile. The set of questions focused on different subjective factors, such as background related to working activities (the role in the development team), the degree of experience in reviewing and writing object-oriented code, and their knowledge of design smell. The experience factor is classified into three categories: fresh (zero years), junior (1 to 3 years), and senior (more than three years). On the other hand, the second part involves a set of classes to be analyzed by the human evaluators if it is considered a god class or includes other smell types. For each class, we included a link that assists the evaluator in showing the source code online. Also, we had a link that indicates the god class definition if the evaluator requires more information. Finally, to reach the most significant number of evaluators with different profiles, we published the designed survey on software companies and software repositories, such as GitHub and SourceForge, LinkedIn, ResearchGate, and forums interested in object-oriented and software refactoring. The survey is available on the replication package [42].

4. Methodology

This section focused on explaining the methodology we follow using the DA algorithm to determine the optimal or near-optimal human evaluators to detect the god class design smell. As can be seen in Fig. 3, the methodology involves two main stages: data preparation and searching for an optimal or near-optimal evaluator profile(s).

Stage 1: Dataset Preparation. Several tools have been proposed for design smell detection. Some tools are designed to be open-source or commercial and work as standalone or plugins. In this work, the source code of the target system was analyzed using the set mentioned in Section 3.1 All tools are available and standard in god class detection. Therefore, it is normal in this context to detect different lists of god classes based on each tool. For this purpose, we merged the whole lists into one list and discarded the duplicated inputs (same god class detected by more than one tool). Afterwards, we assign a weight that ranges from 5 to 1 for each class in the merged list based on the set of tools that detected the god class. For example, class X in rank number one, if detected by the five tools, class Y in rank number two, if detected by the four tools, and so on, until rank number five. However, as manual validation is time-consuming, it is essential. Therefore, we decided to select a random sample of five god classes from the set that were detected by all tools as a god class (weight of class equal to 5) to be evaluated by the human experts in the survey. We adopted the random sample criterion in order to reduce the influence of any potential factor on the results and confirm the high internal validity of the study. Also, selecting the second criterion (weight of class equal to 5) to ensure that the candidate classes selected for manual validation are the true-positive god classes according to the automatic detection.

As mentioned in Section 3.5, each evaluator was asked to fill in some information about his profile and evaluate the set of classes if it is considered a god class or not god class. To perform the optimization process, the dataset requires a specific format. For this purpose, each evaluator in the dataset is represented by a row that includes the following information: reviewing OO code, writing OO code, dealing with design smells, software domain, programming languages, the role in the development team, and the result of god class detection.

Stage 2: Searching for an optimal or near-optimal evaluator profile.

To select the fitted evaluators, the DA is employed, as shown in Fig. 3. The number of initial evaluators will be created, and then used as input values to the stochastic operations in the DA to find the required profiles as follows: According to the number of lines of code of the software system, the number of evaluators is identified by the DA; for instance, the decision-maker will associate different numbers of evaluators for different ranges of the number of lines of code, such as from 1000 to 4000 lines of code a group of 2 evaluators is required. Based on this value, DA will look at each evaluators profile to match those most

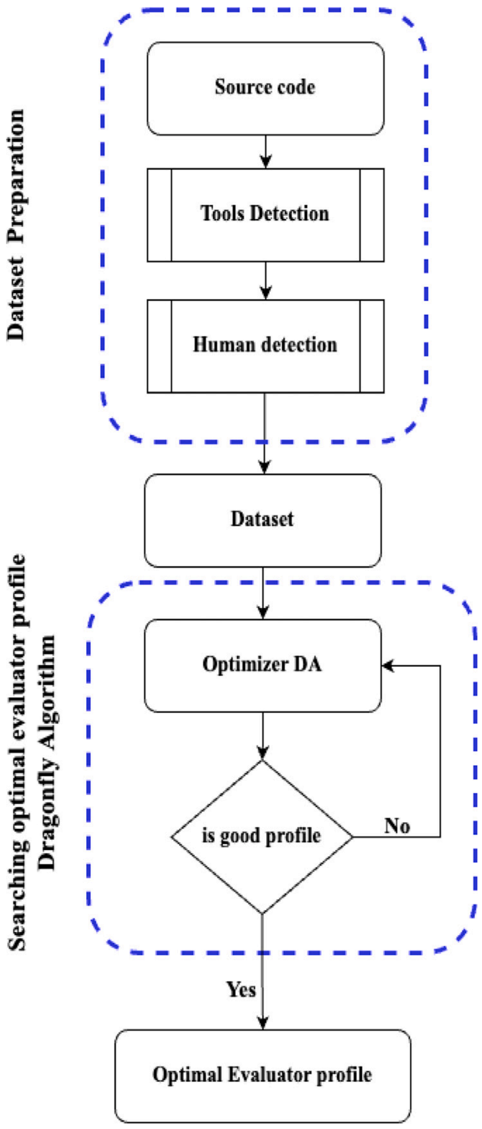


Fig. 3. Proposed approach.

fitted to the required aspects of the software system. The process of approximating for the most fitted evaluators will look at each evaluator in the development role, software development domain, programming languages, experience, and time availability, which should match the details of the software system under investigation. Therefore, the DA will provide at the end a list of those who are suitable for evaluating the software system.

5. Result and discussion

5.1. God class detection

A set of 167 god classes have been detected using detection tools in the target system after discarding the repetitions (merged list). Table 2 shows the number of detected god classes using each tool. For example, JDeodorant detected the highest number of god classes (117), while the lowest was 17 using Together, and this result is expected due to is the only commercial tool. The difference in detection results denotes the low degree of agreement between them if they exist. On the other hand, Table 3 shows the list of god classes detected by the five tools, in which only 6% (10 out of 167) of the whole list. From this list, only

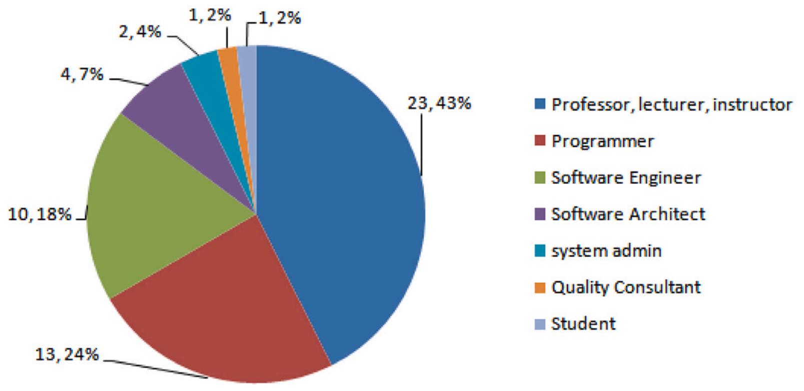


Fig. 4. Distribution of respondents (#,%) over working activities.

Table 2
God class detection results per tool.

Tool	iPlasma	JDeodorant	Together	Decor	PMD
# God class	28	117	17	70	36

Table 3
List of god class detected using the whole number of tools.

#	Class name
1	`ganttproject.chart.ChartModel`
2	`ganttproject.GanttGraphicArea`
3	`ganttproject.GanttGraphicArea.Ta`
4	`ganttproject.GanttOptions`
5	`ganttproject.GanttProject`
6	`ganttproject.GanttResourcePa`
7	`ganttproject.GanttTree2`
8	`ganttproject.GanttTreeTable`
9	`ganttproject.gui.options.CSVS`
10	`ganttproject.ResourceTreeTab`

five classes were chosen randomly and included in the survey to be evaluated by human experts.

5.2. Survey analysis

Descriptive Analysis: According to the designed survey, the total number of respondents was 54 who knew java programming language. In addition, the respondents classified the five classes as having god classes or not. Figs. 4 and 5 show the number and percentage of respondents over the working activities, types, and levels of experience, respectively.

As can be seen in Fig. 4, we can classify the respondents into two groups academic and industry, and we did not find any response that indicates that the evaluator came from both categories. The academic includes the categories of professor, teacher, instructor, and student with a total number of 24 respondents in which the highest number was from the category of professor, teacher, instructor. On the other hand, the industry group involves the categories: of programmer, Software Engineer, Software Architect, Quality Consultant, and system admin, where the total number of respondents was 30. Fig. 5 presents the distribution of numbers and percentage of respondents based on their levels of experience regarding the ability to review object-oriented code(top left), write object-oriented code (top right), and their knowledge of design smells (bottom center). Most of the respondents were from the junior level (1 to 3 years of experience) in all types of experiences.

5.3. Optimization approach

Experiment Settings: The experiments on our approach are performed using MATLAB R2018a based on the code presented by [57]. To allow the determination of preciseness and accuracy, the approach was executed 100 times. To this aim, both mean and standard deviation values are used, by which stability of the solution to the objective function can be determined too. Moreover, the system used to test our approach combines Intel® Core™ i5-9400 CPU with 8 GB RAM memory. The adopted optimization settings for the experiments are motivated by [57] and represented in the Table 4:

Table 4
DA parameter settings.

DA Parameter	Value
Number of iterations	500
No of features	54
No of Agents (population size)	100

It can be seen in Table 4 that the number of iterations by which the optimizer stops searching for new solutions is 500. This value is justified by [57] to perform similarly to the evolutionary algorithms. The number of features is set to 54 according to the data used for experimentations with 54 evaluators' profiles. In addition, the number of search agents or population size used on our problem is 100, as advised by [57,60]. This value represents the initial solutions from which the algorithm will stochastically search for the most fitted ones.

Experiment Results:

To capture the performance of our approach, three outcomes associated with the solution provided on each run are recorded. These outcomes are the computation time consumed by the algorithm to give each solution, the evaluators selected by the algorithm, and the estimated evaluation time span according to the chosen evaluators' productivity. A glimpse of how the algorithm performed in finding an optimal or near-optimal solution for 100 runs in terms of computation time is provided in Fig. 6.

It can be seen in Fig. 6 that the algorithm initially took 18 s to find an optimal or near-optimal solution. Later, the computation time started to stabilize between 15.5 to 16 s. This is due to the nature of heuristic algorithms in which the approximation process moves further on searching for a global optimum. For a problem that consists of 54 evaluators, the computation time on average took a reasonable 15.65 s with a standard deviation of 0.26 to search for the most fitted ones.

However, it is worth investigating whether, at the expense of this computation time, the algorithm will provide precise and accurate solutions or not. Therefore, two criteria are used to capture the accuracy and preciseness of solutions. First, it is essential to see how many instances the algorithm has provided with the least estimated evaluation time span across 100 runs. In addition, if the values of these instances are precise, they are very close to each other. Secondly, if those evaluators selected by the algorithm according to the fitness function have the most productivity needed.

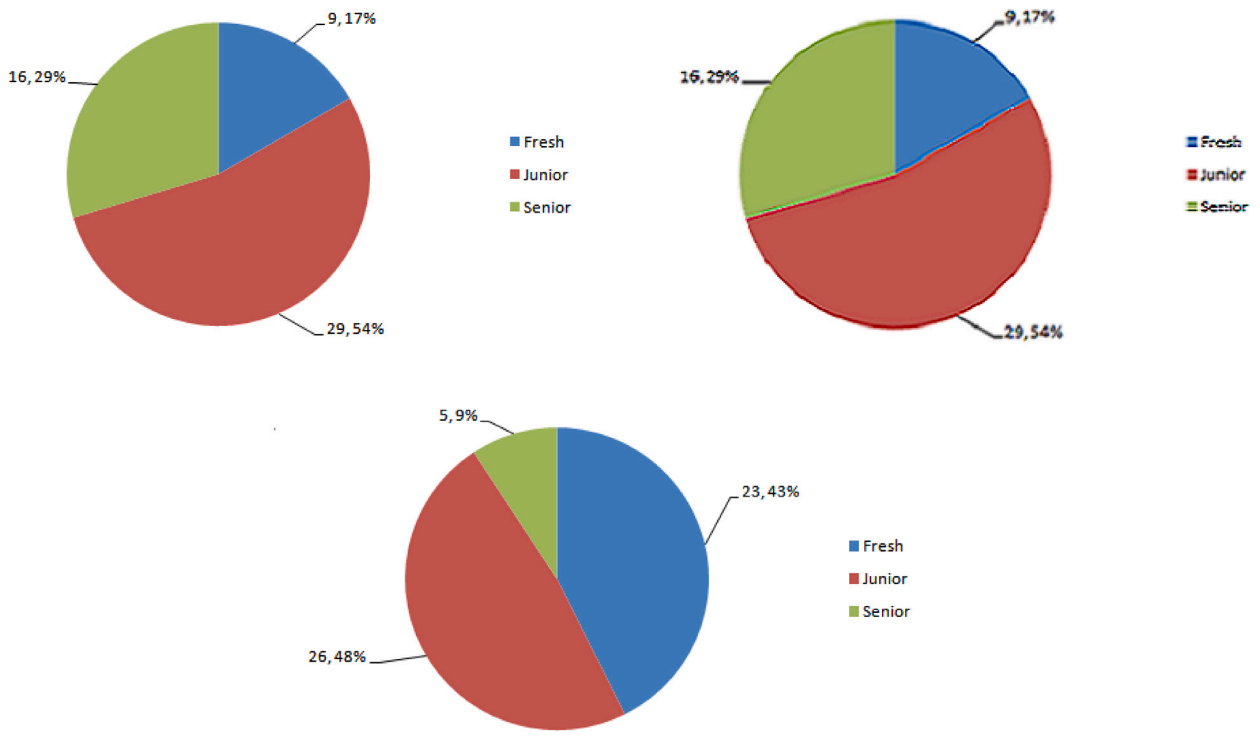


Fig. 5. Left panel: Distribution of respondents (#,%) over level of experience in reviewing Object-oriented code. Right panel: Distribution of respondents (#,%) over level of experience in writing Object-oriented code. Bottom: Distribution of respondents (#,%) over level of experience in code smell.

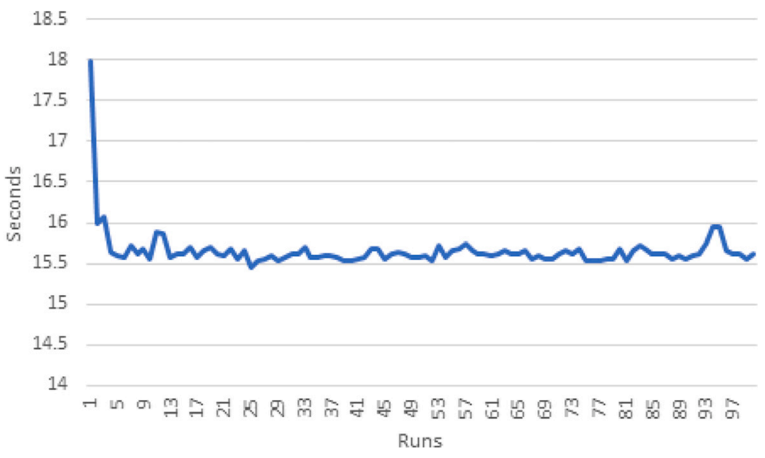


Fig. 6. Computation Time consumed by the algorithm to provide each solution.

These two criteria are correlated in that the least estimated time by the objective function requires involving all evaluators with the highest productivity. At the same time, the evaluators profiles should match the domain and programming languages the evaluators needed for the project. To see whether the algorithm has provided accurate and precise solutions, results of the objective function for 100 runs in terms of estimated evaluation time span are provided in Fig. 7:

Fig. 7 demonstrates how the algorithm is capable of providing, in many instances, the least estimated evaluation time span with an average of 146.45 min for 100 runs. Moreover, we can see from Fig. 7 that only two points representing two estimates have higher values than the rest. Yet, the second-highest point in the figure above illustrates two different solutions. These two solutions provided an estimated evaluation time span of 150.15 min, and the highest provided an estimate of 151.64 min. From this, we can say that only 3% of solutions failed to provide an optimal or near-optimal solution. In contrast, a high percentage of the preciseness of the approach is reached with 97% of solutions that have the exact estimate of 146.32 min.

Now, we need to see how productive the selected evaluators by the algorithm for these solutions are to reach the least estimated evaluation time span. For 100 runs, we can argue that each solution consists of a set of competent evaluators who can perform the evaluation job in a short time span. Nonetheless, we should remember that across the evaluators' profiles, some have a domain, role, and programming language competencies that can support their evaluation task according to the project description and do not match the competencies required for the project.

According to the outcomes of the sets that represent the selected evaluators in the solutions, we found that these solutions mainly consist of 31 evaluators out of 54, and only 3 of those that provide higher estimate values, as Fig. 7 depicts, incorporate 30 evaluators. On the one hand, this represents why these three solutions have a higher estimate, with only 30 evaluators selected. Therefore, choosing one

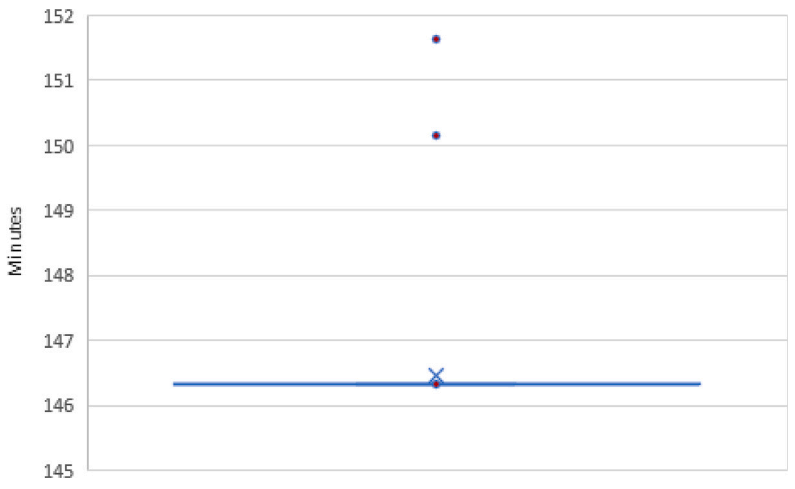


Fig. 7. Results of estimated evaluation time span for 100 runs.

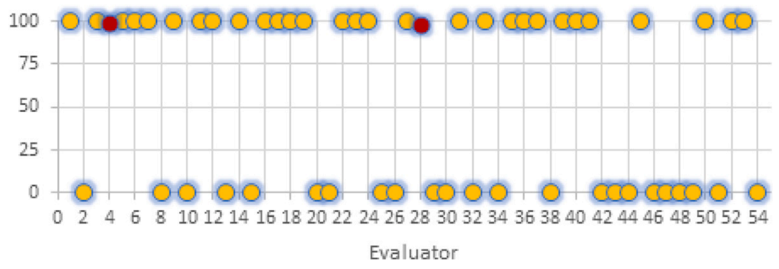


Fig. 8. Frequency of evaluators. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

more evaluator can reduce the time by almost 5 min. For a closer look at the chosen evaluators, we provide in the Fig. 8 the frequency of how many times each evaluator has been selected across 100 runs:

It can be seen in Fig. 8 that 23 evaluators, who are represented by the lower set of yellow dots, have never been selected by any of the solutions. The reason behind this is due to the mismatching of their domain competency with what is required for the project. On the other hand, the upper set of dots consists of 31 evaluators, which the algorithm incorporates to provide the solutions. This set involves the academic (16 evaluators) and the industry (15 evaluators) sectors. The profile characteristics of those evaluators regarding experience levels in reviewing and writing object-oriented code and experience with code smell are shown in Fig. 9. Most of the evaluators have junior levels. This distribution might reflect the real situation of the available experiences of human working activities in this sector. However, evaluators numbers 4 and 28, represented in Fig. 8 by red dots, have been selected 99 and 98 times over 100. Evaluator number 4, for example, has been excluded in run number 54, whereas evaluator number 28 has been excluded in runs 13 and 39. Here we will find that these three runs are those in which the higher estimates are provided as in Fig. 7.

What can be observed is the percentage of agreement of those selected 31 evaluators on detecting god class for the five classes subjects of investigation. Given their level of experience shown in Fig. 9, however, 58% of those evaluators were capable of detecting the first class as a God class. Moreover, 55% among those evaluators have answered the second class as a god class. Then gradually, 61%, 84%, and 100% of those evaluators have stated that god class is detected for the third, fourth, and fifth classes, respectively. These percentages indicate an improvement in the degree of agreement among those evaluators selected by the DA algorithm on detecting god classes. In addition, similarities have been exposed between the profiles of the selected evaluators that may contribute to reducing technical debt and future maintenance costs. Moreover, there have been some responses that incorporated details of god classes, as to some extent, they detected in addition to god class, a partial feature envy, long methods, and code duplication. Noteworthy is that with respect to the overall answers from the 54 evaluators, there have been some responses that identified other types or lacked the identification of a smell to those five classes. Many of those are neglected in the optimized selection by the DA. This can be counted as an improvement toward the minimization of the number of evaluators in the evaluation process.

Table 5
Kappa-Fleiss interpretation.

Kappa-Fleiss value	Interpretation
$0.01 \leq Kappa < 0.20$	Slight
$0.21 \leq Kappa < 0.40$	Fair
$0.41 \leq Kappa < 0.60$	Moderate
$0.61 \leq Kappa < 0.80$	Substantial
$0.81 \leq Kappa \leq 1.00$	Perfect

5.4. Analyzing of agreement between human evaluators

To validate the proposed approach, the Kappa-Fleiss test [61] in R language has been used for computing the degree of agreement between human experts. Kappa results range from 0 to 1, in which if the value is 1, the degree of agreement is perfect. Table 5 shows the interpretation of kappa values.

The degree of agreement among the whole human evaluators (54 respondents) who participated in the survey obtained a kappa value of (0.11). According to Table 5, the kappa value interpretation shows a slight agreement between evaluators. On the other hand, After optimizing the selection of human evaluators using the dragonfly algorithm, only 31 evaluators were selected. As a result, the kappa value is (0.10), which shows a slight agreement between human evaluators on god class detection.

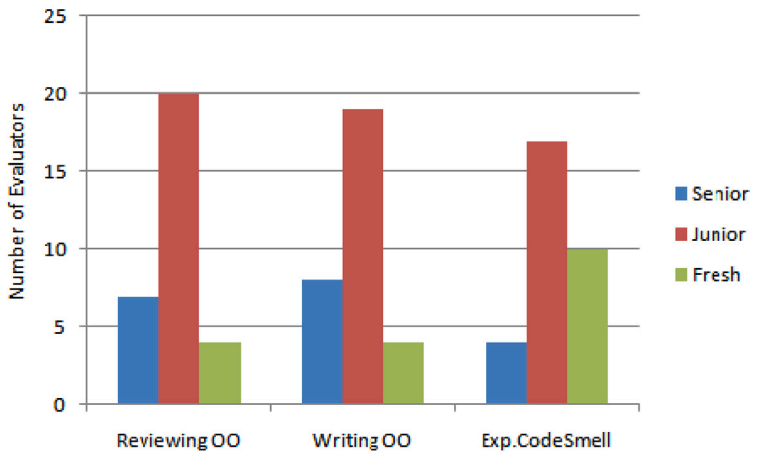


Fig. 9. Characteristics of the selected evaluators profile.

Even though the kappa value of (0.10) for the optimized number of evaluators is less than the one before the optimization with (0.11), both values are very close, with a difference only with (0.01). The obtained findings correspond to the results of previous works regarding the degree of agreement between human evaluators on design smell detection, especially the god class. Consequently, our proposed approach minimizes the maintainability efforts regarding the number of qualified human evaluators involved in god class design smell detection. Moreover, it sheds light on the required time for chosen evaluators by the optimized approach to detect the god class over the selected sample of classes.

6. Threats to validity

Construct validity The main threat to construct validity is the selection of only one design smell "god class" to conduct the study. To overcome this threat, we focused on choosing the design smell that took the attention of the research community. In the literature, god class is the most design smell according to [3]. Also, god class is one of the smells which are common among detection tools. Another threat concerns the group of tools used to detect the god class. A wide set of detection tools in the literature. We overcome this issue by putting rigorous constraints to select the tools, such as analyzing Java source code, common in god class detection, and having high detection precision. **External validity** The significant threat to external validity comes from the nature of the target software analyzed. The software was open-source and implemented in Java. However, to manage this threat, we selected the software system "GanttChart" which is one of the most known systems in the design smell detection context. Especially, the selected version (2.0.10) according to [3]. Therefore, we can generalize the results to open-source software systems written in Java. Also, this generalization is limited because of the selected nature of the design smell and the set of tools used to detect it. **Internal validity** The evaluators reliability is the main threat to internal validity due to the survey is published online. Thus, we have not controlled the survey. We designed the survey in such a way as to decrease the amount of time required to evaluate all the selected classes without boring them. **Conclusion validity** related to all points that influence extracting the right conclusion, such as the dataset collection, detection tools, and human evaluators. All these points have been explained in the stages of the study.

7. Conclusion

Design smell detection is one activity contributing to improving software quality. Several approaches have been proposed for this purpose. Unfortunately, the human context has not been exploited efficiently in domain development. The selection of suitable and qualified human evaluators to perform the design smell detection is a complex problem, given the limitation of resources, variation of development roles and skill availability. This paper introduced an empirical and search-based approach to tackle this problem. A comparison is also presented to provide evidence of whether the degree of agreement is reached between the proposed approach and what the empirical study has provided. By the obtained results, both the empirical study and optimized method have provided relatively similar outcomes. Despite these outcomes showing poor agreement on god class detection, the proposed approach displays the capability of search-based algorithms to minimize the time and effort of design smell detection by selecting the most suitable human evaluators to perform the job. Considering the proportional relationship between time span and cost of design smell detection, the cost in return will be decreased. As per the proposition in this work, the results are not fully generalizable. However, the findings are encouraging for the capability of search-based algorithms to minimize the time and effort of design smell detection by choosing the required human evaluator profile, regardless of the type of design smell we aim to find.

In the future, we plan to extend this study further to establish a relevance mapping between different programming languages that can be converted into weighting scores for the selection of evaluators. This should empirically be performed, having evaluators from different domains or their expertise are in different programming languages to observe their effects on the selection process. This will be done by performing many experiments according to the mappings to show how the selection of evaluators can be impacted by these factors. To extend this study further and to improve the results, we also will replicate the work by including other subjective factors, such as region, and programming language, involving more design smells, and increasing the sample of classes for evaluation by human experts. Moreover, we plan to employ different search-based algorithms to allow for suitability and capacity comparison in solving the selection of human evaluator-based design smell detection.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Sadi Alawadi reports financial support was provided by Uppsala University.

Data availability

The data that has been used is confidential.

References

[1] F. Pérez, Refactoring Planning for Design Smell Correction in Object-Oriented Software (Ph.D. thesis), School of Engineering, Valladolid University, 2011.

[2] W.H. Brown, R.C. Malveau, H.W. McCormick, T.J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, John Wiley & Sons, Inc., 1998.

[3] K. Alkharabsheh, Y. Crespo, E. Manso, J. Taboada, Software design smell detection: a systematic mapping study, Softw. Qual. J. (2018).

[4] P.F. Mihancea, R. Marinescu, Towards the optimization of automatic detection of design flaws in object-oriented software systems, in: Ninth European Conference on Software Maintenance and Reengineering, 2005, pp. 92–101, http://dx.doi.org/10.1109/CSMR.2005.53.

[5] U. Azadi, F.A. Fontana, M. Zanoni, Poster: Machine learning based code smell detection through WekaNose, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18, ACM, 2018, pp. 288–289, http://dx.doi.org/10.1145/3183440.3194974.

[6] F.A. Fontana, P. Braione, M. Zanoni, Automatic detection of bad smells in code: An experimental assessment, J. Object Technol. 11 (2) (2012) 5:1–38.

[7] M. Mäntylä, J. Vanhanen, C. Lassenius, Bad smells - humans as code critics, in: 20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA, 2004, pp. 399–408.

[8] M. Mäntylä, C. Lassenius, Subjective evaluation of software evolvability using code smells: An empirical study, Empir. Softw. Eng. 11 (3) (2006) 395–431.

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

[10] A.F. Yamashita, L. Moonen, Do developers care about code smells? An exploratory survey, in: 20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013, 2013, pp. 242–251.

[11] 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.

[12] K. Alkharabsheh, Y. Crespo, J.Á. TAboada, M. Esperanza, Sobre el grado de acuerdo entre evaluadores en la detección de design smells, in: XXI Jornadas de IngenierÍa de Software Y Bases de Datos, 2016, pp. 1–15.

[13] M. Choinzon, Y. Ueda, Detecting defects in object oriented designs using design metrics, in: J. Conf. on Knowledge-Based Software Engineering, 2006, pp. 61–72.

[14] R. Fourati, N. Bouassida, H. Abdallah, A metric-based approach for anti-pattern detection in UML designs, Comput. Inf. Sci. (2011) 17–33.

[15] C. Marinescu, R. Marinescu, P.F. Mihancea, R. Wettel, iPlasma: An integrated platform for quality assessment of object-oriented design, in: Intl. Conf. Software Maintenance - Industrial and Tool Volume, 2005, pp. 77–80.

[16] N. Moha, Detection and correction of design defects in object-oriented designs, in: Conf. on Object-Oriented Programming Systems and Applications Companion, 2007, pp. 949–950.

[17] N. Moha, Y.-G. Guéhéneuc, DECOR: a tool for the detection of design defects, in: Intl. Conf. on Automated Software Engineering, 2007, pp. 527–528.

[18] K. Alkharabsheh, Y. Crespo, M. Fernández-Delgado, J.M. Cotos, J.A. Taboada, Assessing the influence of size category of the project in god class detection, an experimental approach based on machine learning (MLA), in: International Conference on Software Engineering & Knowledge Engineering, 2019, pp. 361–366.

[19] M.J. Munro, Product metrics for automatic identification of "bad smell" design problems in java source-code, in: Intl. Conf. Software Metrics, 2005, pp. 15–15.

[20] R. Shatnawi, Deriving metrics thresholds using log transformation, J. Softw.: Evol. Process 27 (2) (2015) 95–113.

[21] L. Tahvildar, K. Kontogiannis, Improving design quality using meta-pattern transformations: a metric-based approach, J. Softw.: Evol. Process 16 (4–5) (2004) 331–361.

[22] S. Hassaine, F. Khomh, Y.-G. Guéhéneuc, S. Hamel, IDS: an immune-inspired approach for the detection of software design smells, in: Intl. Conf. Quality of Information and Communications Technology, 2010, pp. 343–348.

[23] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, BDTEX: A GQM-based Bayesian approach for the detection of antipatterns, J. Syst. Softw. 84 (4) (2011) 559–572.

[24] J. Kreimer, Adaptive detection of design flaws, Electron. Notes Theor. Comput. Sci. 141 (4) (2005) 117–136.

[25] N. Maneerat, P. Muenchaisri, Bad-smell prediction from software design model using machine learning techniques, in: Intl. J. Conf. on Computer Science and Software Engineering, 2011, pp. 331–336.

[26] K. Alkharabsheh, S. Alawadi, V.R. Kebande, Y. Crespo, M. Fernández-Delgado, J.A. Taboada, A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of god class, Inf. Softw. Technol. 143 (2022) 106736.

[27] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Jdeodorant: Identification and removal of type-checking bad smells, in: Intl. Conf. on Software Maintenance and Reengineering, 2008, pp. 329–331.

[28] F.A. Fontana, E. Mariani, A. Morniroli, R. Sormani, A. Tonello, An experience report on using code smells detection tools, in: Fourth International IEEE Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings, 2011, pp. 450–457.

[29] F.A. Fontana, P. Braione, M. Zanoni, Automatic detection of bad smells in code: An experimental assessment, J. Object Technol. 11 (2) (2012) 5: 1–38.

[30] A. Hamid, M. Ilyas, M. Hummayun, A. Nawaz, A comparative study on code smell detection tools, Int. J. Adv. Sci. Technol. 60 (2013) 25–32.

[31] G. Rasool, Z. Arshad, A review of code smell mining techniques, J. Softw. Evol. Process 27 (11) (2015) 867–895.

[32] 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, 2016, pp. 159–172.

[33] M.R. Chaudron, B. Katumba, X. Ran, Automated prioritization of metrics-based design flaws in UML class diagrams, in: Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on, IEEE, 2014, pp. 369–376.

[34] S.A. Vidal, C. Marcos, J.A. Díaz-Pace, An approach to prioritize code smells for refactoring, Autom. Softw. Eng. 23 (3) (2016) 501–532.

[35] R. Arcoverde, E. Guimarães, I. Macía, A. Garcia, Y. Cai, Prioritization of code anomalies based on architecture sensitiveness, in: Software Engineering (SBES), 2013 27th Brazilian Symposium on, IEEE, 2013, pp. 69–78.

[36] A. Ghannem, M. Kessentini, G. El Boussaidi, Detecting model refactoring opportunities using heuristic search, in: Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '11, IBM Corp., USA, 2011, pp. 175–187.

[37] A. Ouni, M. Kessentini, K. Inoue, M. Cinnéide, Search-based web service antipatterns detection, IEEE Trans. Serv. Comput. 10 (4) (2017) 603–617, http://dx.doi.org/10.1109/TSC.2015.2502595.

[38] S. Fu, B. Shen, Code bad smell detection through evolutionary data mining, in: 2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2015, pp. 1–9, http://dx.doi.org/10.1109/ESEM.2015.7321194.

[39] G. Saranya, H. Khanna Nehemiah, A. Kannan, V. Nithya, Model level code smell detection using EGAPSO based on similarity measures, Alexandria Eng. J. 57 (3) (2018) 1631–1642.

[40] M. Draz, M. Farhan, S. Abdulkader, M. Gafar, Code smell detection using whale optimization algorithm, Comput. Mater. Continua 68 (2) (2021) 1919–1935.

[41] K. Alkharabsheh, Y. Crespo, M. Fernandez-Delgado, J. Viqueira, J. Taboada, Exploratory study of the impact of project domain and size category on the detection of the god class design smell, Softw. Qual. J. (2021).

[42] k. Alkharabsheh, S. Al khatib, Replication package of raw data, scripts and all necessary material for replication, 2022, URL: https://drive.google.com/drive/folders/1tpFZatlNQkjHNrvDAPNgcFTulntQeYPk.

[43] K. Alkharabsheh, S. Alawadi, Y. Crespo, M.E. Manso, J.A.T. González, Analysing agreement among different evaluators in god class and feature envy detection, IEEE Access 9 (2021) 145191–145211.

[44] A. Tahir, J. Dietrich, S. Counsell, S. Licorish, A. Yamashita, A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites, Inf. Softw. Technol. 125 (2020) 106333.

[45] R. Oliveira, R. de Mello, E. Fernandes, A. Garcia, C. Lucena, Collaborative or individual identification of code smells? On the effectiveness of novice and professional developers, Inf. Softw. Technol. 120 (2020) 106242.

[46] J. Martins, C. Bezerra, A. Uchôa, A. Garcia, How do code smell co-occurrences removal impact internal quality attributes? A developers' perspective, in: Brazilian Symposium on Software Engineering, 2021, pp. 54–63.

[47] X. Han, A. Tahir, P. Liang, S. Counsell, Y. Luo, Understanding code smell detection via code review: A study of the openstack community, in: 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC), IEEE, 2021, pp. 323–334.

[48] J.A. Santos, M.G. de Mendonça, C.V. Silva, An exploratory study to investigate the impact of conceptualization in god class detection, in: Intl. Conf. on Evaluation and Assessment in Software Engineering, 2013, pp. 48–59.

[49] N. Anquetil, A. Etien, G. Andreo, S. Ducasse, Decomposing god classes at siemens, in: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2019, pp. 169–180.

[50] K. Alkharabsheh, An empirical study on the co-occurrence of design smells in the same software module:God class case study, in: 2021 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT), 2021, pp. 1–6, http://dx.doi.org/10.1109/JEEIT53412.2021.9634144.

[51] D.I. Sjøberg, A. Yamashita, B.C. Anda, A. Mockus, T. Dybå, Quantifying the effect of code smells on maintenance effort, IEEE Trans. Softw. Eng. 39 (8) (2012) 1144–1156.

[52] Z. Soh, A. Yamashita, F. Khomh, Y.-G. Guéhéneuc, Do code smells impact the effort of different maintenance programming activities? in: 2016 IEEE 23Rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, IEEE, 2016, pp. 393–402.

[53] M. Fowler, K. Beck, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999.

[54] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, Springer Science & Business Media, 2007.
[55] M. Hitz, B. Montazeri, Chidamber and kemerersḿetrics suite: A measurement theory perspective, IEEE Trans. Softw. Eng. 22 (04) (1996) 267–271.
[56] B. Henderson-Sellers, The mathematical validity of software metrics, ACM SIGSOFT Softw. Eng. Not. 21 (5) (1996) 89–94.
[57] S. Mirjalili, Dragonfly algorithm: a new meta-heuristic optimization technique for solving single-objective, discrete, and multi-objective problems, Neural Comput. Appl. 27 (4) (2016) 1053–1073.
[58] S. Al Khatib, A Comparative Study of the Relative Performance and Real-World Suitability of Optimization Approaches for Human Resource Allocation (Ph.D. thesis), University of East Anglia, 2018.
[59] S.M. Al Khatib, J. Noppen, Benchmarking and comparison of software project human resource allocation optimization approaches, ACM SIGSOFT Softw. Eng. Not. 41 (6) (2017) 1–6.
[60] S.M. Al Khatib, Optimization of path selection and code-coverage in regression testing using dragonfly algorithm, in: 2021 International Conference on Information Technology (ICIT), IEEE, 2021, pp. 919–923.
[61] K. Gwet, Handbook of Inter-Rater Reliability: the Definitive Guide to Measuring the Extent of Agreement Among Raters, fourth ed., 2012, pp. 73–100.