A Position-Aware Approach to Decomposing God Classes

Tianyi Chen*
Beijing Institute of Technology
Beijing, China
chentianyiyx@gmail.com

Yanjie Jiang*
Peking University
Beijing, China
yanjiejiang@pku.edu.cn

Fu Fan
Beijing Institute of Technology
Beijing, China
fufan@bit.edu.cn

Bo Liu
Beijing Institute of Technology
Beijing, China
liubo@bit.edu.cn

Hui Liu†
Beijing Institute of Technology
Beijing, China
liuhui08@bit.edu.cn

ABSTRACT

God classes are widely recognized as code smells, significantly impairing the maintainability and readability of source code. However, resolving the identified God classes remains a formidable challenge, and we still lack automated and accurate tools to resolve God classes automatically. To this end, in this paper, we propose a novel approach (called *ClassSplitter*) to decompose God classes. The key observation behind the proposed approach is that software entities (i.e., methods and fields) that are physically adjacent often have strong semantic correlations and thus have a great chance of being classified into the same class during God class deposition. We validate this hypothesis by analyzing 54 God class decomposition refactorings actually conducted in the wild. According to the observation, we measure the similarity between software entities by exploiting not only traditional code metrics but also their relative physical positions. Based on the similarity, we customize a clustering algorithm to classify the methods within a given God class, and each of the resulting clusters is taken as a new class. Finally, ClassSplitter allocates the fields of the God class to the new classes according to the field-access-based coupling between fields and classes. We evaluate ClassSplitter using 133 real-world God classes from open-source applications. Our evaluation results suggest that ClassSplitter could substantially improve the state of the art in God class decomposition, improving the average MoJoFM by 47%. Manual evaluation also confirmed that in most cases (77%) the solutions suggested by ClassSplitter were preferred by developers to alternatives suggested by the state-of-the-art baseline approach.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; **Software evolution**.

*Tianyi Chen and Yanjie Jiang made equal contributions to this work.
†Corresponding author

KEYWORDS

God Class, Software Refactoring, Code Smells, Large Language Model

ACM Reference Format:
Tianyi Chen, Yanjie Jiang, Fu Fan, Bo Liu, and Hui Liu. 2024. A Position-Aware Approach to Decomposing God Classes. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3691620.3694992

1 INTRODUCTION

Software refactoring is a widely-used technique to improve software quality, especially its readability and maintainability [34] [36]. The key to software refactoring is to restructure the internal structures of the software applications, e.g., relocating code elements like methods, fields, classes, and packages. To identify which part of the source code deserves refactorings, Beck and Fowler [9] proposed the concept of *bad smells*. According to the definition, bad smells are "*certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring*" [9]. Consequently, by identifying bad smells, we may identify refactoring opportunities.

God class is a typical bad smell [9]. God classes often refer to such huge classes that do too much, taking a lot of responsibilities (like the almighty God). God classes are harmful because they violate the single-responsibility principle [33], a well-known principle that requests each module to take only a single responsibility. God class, in contrast, takes multiple responsibilities, which may result in a sequence of negative impacts on the maintainability of the source code. On the first place, it is difficult to read and understand the God class because of its complexity. Second, the complexity of the God class also makes it difficult to be reused. In case you need only a small part of the class, you would hesitate to reuse the whole (complex) class because the latter would bring useless code as well as unnecessary coupling (with external elements) and unnecessary complexity. Finally, since a God class takes multiple responsibilities, it could be modified frequently because of various reasons associated with various responsibilities. The modification is not only challenging (because of the complexity of the class) but also makes the God class unstable. Because of all such negative impacts, developers had better detect and decompose God classes.

However, it is often challenging to decompose a God class correctly into smaller classes [41]. Various functions fulfilled by diffident methods often intertwine closely, making it difficult to cut them off clearly. Besides that, it is difficult, if not impossible, to quantitatively measure the quality of different decomposition solutions. Consequently, it is challenging to select the best solution by comparing all potential solutions. To facilitate the task, approaches have been proposed to decompose God classes automatically or semi-automatically. For example, Bavota et al. [7] proposed an approach to decompose God classes according to the similarity between software entities. The similarity between the two entities concerns their common access to other elements (e.g., fields), caller-callee relationships, and their textual similarity computed by LSI (Latent Semantic Indexing). Akash et al. [1] improved Bavota's approach by replacing LSI with LDA (Linear Discriminant Analysis) in the computation of the textual similarity between methods. Anquetil et al. [3] designed a software visualization tool and an accompanying process that together allow programmers to conveniently design decomposition solutions to God classes. The approach proposed by Alzahrani [2] exploits code metrics only, and it leverages a greedy algorithm to cluster methods according to the code metrics. Although such approaches are useful and they successfully reduce the cost of God class decomposition, the performance of such approaches deserves further improvement (as shown in Section 5). Developers often have to substantially modify the suggested solutions.

To this end, in this paper, we propose a novel approach (called *ClassSplitter*) to decompose God classes. The key observation behind the approach is that the relative positions (i.e., the physical orders) of fields and methods within a God class are useful in suggesting decomposition solutions. According to our experience, physically adjacent methods are often closely related and thus have a great chance of being allocated to the same class during God class decomposition. To the best of our knowledge, however, none of the existing approaches have exploited such positions for God class decomposition. Besides the positions, we also exploit traditional code metrics as well as function-based similarity between methods. We leverage a heavily customized clustering algorithm to decompose methods within the given God class into small groups according to function-based similarities, and to adapt (by splitting and merging) the groups according to the methods' positions and code metrics. Finally, we allocate fields to the resulting groups according to the coupling between fields and methods. We evaluate the proposed approach using 133 real-world God classes and compare its solutions against the golden set (i.e., solutions constructed by the original developers). Our evaluation results suggest that the proposed approach can substantially improve the state of the art, improving the average performance (MoJoFM [47]) by 47%. Human evaluation results also confirm that solutions generated by the proposed approach were often (in 77% of the cases) better than those suggested by the state-of-the-art baseline.

This paper makes the following contributions:

- We propose a novel approach to decomposing God classes. To the best of our knowledge, it is the first one in this line that exploits the relative positions of methods and fields.
- The paper presents a new dataset containing 187 real-world God classes and their reference solutions proposed and applied by the original developers. To the best of our knowledge, this is the first publicly available dataset (available at [15]) of God classes accompanied by reference solutions.
- We conduct an empirical study that validates the correlation between the physical positions of methods/fields and their probability of co-occurrence after God class decomposition.
- The paper presents a prototype implementation and an initial evaluation of the proposed approach. The replication package is publicly available on GitHub [15].

2 RELATED WORK

2.1 Decomposition of God Classes

The approach proposed by De Lucia et al. [19] is the first to decompose large (God) classes automatically. They consider both structural and conceptual criteria and build a weighted graph of the class methods based on structural and semantic cohesion metrics. With the metrics, they employ a *MaxFlow-MinCut* algorithm [17] to split the graph to produce more cohesive classes.

Fokaefs et al. [22] employed structural information of software entities and a hierarchical agglomerative clustering algorithm to suggest extract class refactorings. To do that, they first retrieve entity sets (i.e., entities coupled with the given entity) [44] for each entity in the given class, and measure the distance between two class entities by the Jaccard distance[4] between their entity sets. The Jaccard distance is then employed by the clustering algorithm. The algorithm first assigns each entity to a single cluster, and then in each iteration, it merges two closest clusters until all entities are contained in a single cluster. The output of the clustering algorithm is a tree diagram where leaves represent entities to be clustered and other nodes represent possible clusters composed of entities. The root of the tree represents the largest cluster containing all of the entities. Because it is often difficult to find the right level to cut the dendrogram, the authors tried to mitigate this issue by proposing different refactoring opportunities that can be obtained using different thresholds. The approach has also been implemented [23].

Bavota et al. [8] presented a novel approach to separate a God class into two smaller ones. It represents the methods in the to-be-decomposed class as a weighted graph and then employs the *MaxFlow-MinCut* technique [17] to divide it into two subgraphs. The weight between methods (nodes) is computed as the combination of the semantic and structural similarity between methods. The structural similarity is measured by variables accessed by methods and their inter-invocations. The semantic similarity is measured by Latent Semantic Indexing(LSI) [20] according to vocabularies extracted from the methods. To the best of our knowledge, they are the first to exploit such semantic similarities for class decomposition. This approach was later improved by Bavota et al. [7]. The improved version applies a two-step clustering algorithm (instead of *MaxFlow-MinCut*) to separate a large graph into multiple subgraphs (can be more than 2) and to avoid obtaining classes with a very low number of methods.

The approach proposed by Akash et al. [1] is similar to that proposed by Bavota et al. [8]. The difference is that the former exploits the semantic information in methods with Latent Dirichlet Allocation(LDA)[12] instead of LSI that was employed by Bavota et al. [8]. LAD considers the words in each method as a document and gives a topic distribution for each document. The cosine similarity between the topic distributions of the methods serves as their semantic similarity.

Jeba et al. [27] extended Bavota's approach [8] by employing a novel clustering algorithm that exploits documentation of methods. It exploits the comment sections appearing exactly before the method and the comment lines within the method body. Such documents are then normalized and weighted by the term frequency-inverse document frequency (TF-IDF). The similarity between two method documents is calculated as the cosine of the angle between their corresponding vectors. It then employs the Hierarchical Agglomerative Clustering to cluster the methods. It has high requirements for the completeness of comments in code.

Alzahrani [2] proposed a novel approach to decompose classes according to cohesion and coupling. The approach defines quantitative metrics for measuring cohesion and coupling between classes. The most novel part of the approach is that it leverages a greedy clustering algorithm. The algorithm initially takes each method in the to-be-decomposed class as a candidate class (which contains only a single method). It then keeps merging class pairs that have the highest coupling until only two classes are left. The clustering with the best metrics (considering both cohesion and coupling) is recommended as the final solution.

Our approach differs from the existing approaches introduced above in that it exploits the relative positions of software entities that have not yet been exploited by other approaches in this line.

2.2 Detection of God Classes

Automated detection of God classes has been extensively studied, and quite a few approaches have been proposed. We refer to the literature review conducted by Reis et al. [39] for a comprehensive introduction to such approaches. Here, we only present a brief introduction to the most impressive and the most influential approaches.

One of the most intuitive tactics to detect God classes is to define a set of heuristics rules and to validate to-be-checked classes against such rules. We call approaches following such tactic *rule-based approaches*. For example, Simon et al. [43] provided a metric-based software visualization approach that visually presents all elements within a class. The distance between nodes (code entities) is computed by the coincide of the entities they associate with. If the elements composed of several highly cohesive groups and different groups are far away from each other, it is likely that the original class is a God class and it should be decomposed into smaller ones. Marinescu et al. [32] identified God classes with three code metrics: *Weighted Method Count* (measuring the complexity of the class), *Tight Class Cohesion* (measuring the cohesion of the class) and *Access to Foreign Data*(measuring the accesses to foreign data). They computed the metrics and compared them against predefined thresholds to decide whether it was a God class or not. Palomba et al. [37] were the first to detect God classes with source code's change histories. A class is identified as a God class if it has been "*modified in more than α percent of commits involving at least another class*" [37]. Palomba et al. [38] also leveraged textual similarity between software entities to detect God classes. If the average textual similarity between any method pairs within the given class is smaller than a threshold, it is reported as a God class.

With the popularity of machine learning techniques, machine learning-based approaches to God class detection are emerging [14]. For example, Khomh et al. [28, 29] employed Bayesian belief networks (BBN) to God classes whereas Maiga et al. [31] leveraged Support Vector Machines (SVM). Fontana et al. [6] compared various machine learning algorithms in detecting God classes, and their evaluation results suggest that *J48* [10], *JRip* [16], *Naive Bayes*, and *Random Forest* often resulted in the best performance. Fontana et al. [24] not only detected God classes but also leveraged machine learning techniques to predict the severity. Liu et al. [30] applied deep learning techniques to God class detection. Their key contribution is that they proposed a refactoring-based approach to generate training data for smell detection, which makes deep learning-based smell detection feasible. Sharma et al. [42] further improved deep learning-based God class detection with transfer learning [46].

3 HYPOTHESIS AND ITS VALIDATION

3.1 Observation and Hypothesis

While collecting real-world God classes and their refactoring solutions conducted by developers, we found that physically adjacent methods are often closely related in their function, and thus in most cases, they were allocated to the same classes during God class decomposition. A typical example is presented in Listing 1. This example is extracted from the open-source application *Ant Media Server* [5]. The class *StreamSourceRestService* is composed of 25 methods and 8 fields. To improve the readability and maintainability of the source code, the developers of the application decided to decompose it into two smaller classes: *StreamSourceRestService* and *RestServiceBase*. The elements (methods and fields) marked in red are extracted into a new class while the others are kept. From the example, we make the following observations:

- First, three blocks of successively located methods were extracted into the new class. That is, the ten extracted methods are not uniformly distributed across the document, but gathering in three consecutive physical regions (blocks).
- Second, physically adjacent methods were frequently classified into the same classes. Among the 24 pairs of the adjacent methods, only 21%=5/24 were broken by the class decomposition.

To reveal why successively located methods are often semantically related (and are extracted together), we first looked into the evolution history of the class in Listing 1. We observed that the methods had been added in multiple commits. However, for each of the commits, all methods added by this commit were placed in a single successive block. That is, for some special reasons (e.g., implementation of a new feature), a set of closely related methods was created and all such methods were placed in a successive block. As a result, when the God class was decomposed, the whole block (with successive methods) was extracted into the new class.

Based on the observation, we make the following hypotheses:

Hypothesis 1. *Physically adjacent methods/fields are likely to be allocated to the same class during God class decomposition.*

Hypothesis 2. *During God class decomposition, developers often extract method/field blocks (with successive methods/fields) instead of individual methods/fields.*

```
public class StreamsSourceRestService {
    private static final String HTTP
    private ServletContext servletContext
    private DataStoreFactory dataStoreFactory
    private IDataStore dbStore
    private ApplicationContext appCtx
    private IScope scope
    private AntMediaApplicationAdapter appInstance
    protected static Logger logger
    public Result addStreamSource()
    public String getRTSPSteramURI()
    public Result addIPCamera()
    public Result addSource()
    private void addSocialEndpoints()
    public Result getCameraError()
    public Result synchUserVodList()
    public Result updateCamInfo()
    public String[] searchOnvifDevices()
    public Result moveUp()
    public Result moveDown()
    public Result moveLeft()
    public Result moveRight()
    private ApplicationContext getAppContext()
    public AntMediaApplicationAdapter getInstance()
    public IScope getScope()
    public void setScope()
    public IDataStore getStore()
    public void setDataStore()
    public void setCameraStore()
    public boolean validateIPaddress()
    public boolean checkStreamUrl()
    public boolean checkIPCamAddr()
    public DataStoreFactory getDataStoreFactory()
    public void setDataStoreFactory()
}
```

Listing 1: Positions of Methods Extracted During Class Decomposition

3.2 Validation

To validate the hypotheses, we collected a set of real-world God classes as well as their decomposition solutions as follows:

- First, we selected Java projects from Github that were ranked in the top 1,000 based on their stars. We further ranked such projects according to their project names (resulting in a list $Projs$) and tried to discover God classes from them in order.
- Second, from the first project in $Projs$, we leveraged RefactoringMiner [45] to identify God class-related refactorings (i.e., *extract super class*, *extract sub class*, and *split class*) applied to this project, and removed the project from the list $Projs$. RefactoringMiner was used because it represents the state of the art in refactoring discovery.
- Third, we analyzed the reported potential refactorings and finalized a list of manually confirmed God classes as well as their decomposition solutions.
- If the allocated time slot (one week) ran out, the collection terminated. Otherwise, we turned to the second step and continued with the next project.

Notably, not all God class-related refactorings (i.e., *extract super class*, *extract sub class*, and *split class*) are associated with God classes, and thus manual analysis and validation is indispensable. The manual analysis is not only time-consuming but also subjective because we still lack quantitative guidance on the identification of the refactoring intents: These refactorings could be employed for various purposes besides the decomposition of God classes. To make the process as objective and reproducible as possible, according to our initial analysis on the data we got about the false positive in the data, we defined the following objective criterion to exclude false positives: If the split Class has less than 20 methods or less than 20% of its methods have been moved out of it, it is unlikely to be a God class and it is excluded for further manual analysis. Such criteria are empirically to exclude false positives (so as to reduce the cost of manual analysis). There might be a few true positive filtered in this step, but finding them is too costly. For the remaining classes split by the reported refactoring, two authors of the paper manually analyzed the likely purpose of the refactorings. They first analyzed the size of the decomposed classes and the number of newly added features (methods and fields) in the new classes. If the refactorings were conducted to move a small number of features from the original normal-sized class to recently added classes where most of the features were not moved from the original class, it is likely that the original class is not a God class. It just happened that the developers created new classes and some features of the original class were moved to the new classes for some reasons like *feature envy*. The two authors analyzed each of the potential God classes independently. In case of inconsistency, they discussed together, and they managed to reach a consensus on all cases. The kappa coefficient of 0.492 suggests moderate consistency between the participants.

Table 1: Discovered Refactorings for Class Decomposition

Items	Values (Average)
# Extracted Methods	13.65
# Extracted Fields	4.09
# Extracted Method Blocks	2.87
# Extracted Field Blocks	1.31
# Extracted Isolated Methods	0.87
# Extracted Isolated Fields	0.52
# Methods per Extracted Method Block	4.75
# Fields per Extracted Field Block	3.12

The data collection resulted in 54 God classes and their original solutions coming from 25 Java projects. For each God class, we enumerated each pair of the adjacent methods within the God class and validated whether they were allocated to the same class during the God class decomposition. We also analyzed adjacent fields in the same way. Our analysis results validated Hypothesis 1:

- Adjacent methods are much more likely than others to be allocated to the same class during God class decomposition. The possibility for two adjacent methods to be allocated to the same class is 84.7%, substantially higher than that between randomly selected method pairs (58.7%). To rigorously and quantifiably validate the hypothesis, we conducted statistical testing on our data. The low p-value (1.6e-10) from the U test indicates that adjacent methods are significantly more likely to be allocated to the same classes. To validate the universality of the hypothesis, we analyzed our data and found that in 50 out of 54 God classes, adjacent method pairs were at least 20% more likely to

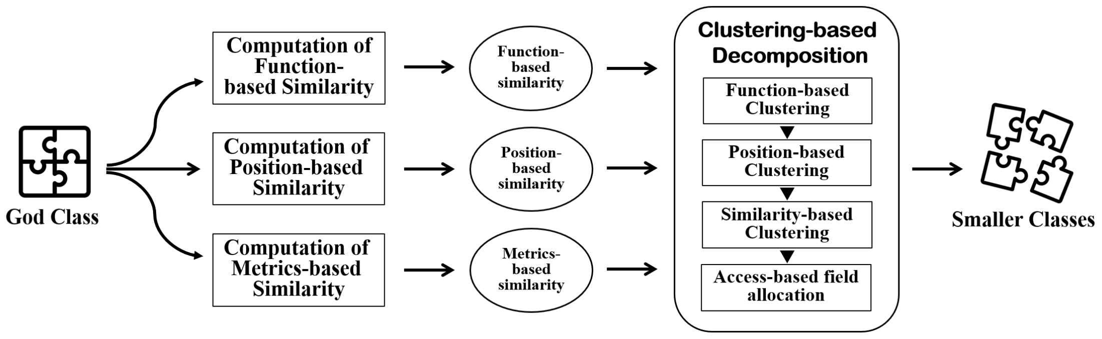
God Class
Computation of Function-based Similarity
Computation of Position-based Similarity
Computation of Metrics-based Similarity
Function-based similarity
Position-based similarity
Metrics-based similarity
Clustering-based Decomposition
Function-based Clustering
Position-based Clustering
Similarity-based Clustering
Access-based field allocation
Smaller Classes

Figure 1: Overview of the Proposed Approach

be allocated to the same classes compared to non-adjacent pairs. This suggests that the influence of position is highly prevalent.
- Adjacent fields also have a great possibility of being allocated to the same class. The average possibility is 83.2%, comparable to that (84.7%) of adjacent methods.

To validate Hypothesis 2, we investigated how methods/fields had been extracted from the 54 analyzed God classes. The results are presented in Table 1 where we make the following observations:
- Methods are often extracted as blocks. On average, 13.65 methods are extracted when an *extract class* refactoring is conducted where 94%=(13.65-0.87)/13.65 of the methods are extracted with their neighbors. Only 6%=0.87/13.65 are extracted as *isolated* individuals, i.e., none of their neighbors are extracted together with them. The same is true for fields. Around 87%=(4.09-0.52)/4.09 of the fields are extracted as blocks (i.e., with their neighbors).
- The size of the extracted blocks is considerable. The average size of method blocks is 4.75, and the average size of field blocks is 3.12. It may suggest that once a method is selected for extraction, we may identify four additional methods around the selected one to be extracted together with it.

4 APPROACH

4.1 Overview

An overview of the proposed approach (*ClassSplitter*) is presented in Fig. 1. Overall, it is composed of two parts. In the first part, it computes three categories of semantic relationships among software entities within the given God class. In the second part, it employs a customized clustering algorithm to classify software entities (i.e., methods and fields) into groups where each group represents a new class. For a give God class $GC =< f_1, f_2 \dots f_n, m_1, m_2 \dots m_k >$ where f_i and m_j represent fields and methods within the class, the proposed approach works as follows:
- It computes the metrics-based similarity (noted as $MS(m_i, m_j)$), the position-based similarity (noted as $PS(m_i, m_j)$), and the function-based similarity (noted as $FS(m_i, m_j)$) between any pair of methods within the God class;
- It initializes some small groups by connecting methods that are highly similar in function, i.e., with great $FS(m_i, m_j)$. The rationale for the initial grouping is that highly similar methods should be classified into the same group (class).
- It adjusts the initial groups according to the methods' positions. If an initial group (of methods) is separated physically by other methods into smaller groups, the proposed approach replaces the initial group with the smaller ones so that all elements within a single group are physically consecutive. If a sequence of small groups (each contains a single method) is physically consecutive, the proposed approach concatenates them as a single group.
- It merges small groups according to their similarity in code metrics, positions, and functions.
- It allocates fields within the God class to the groups (classes) generated in the preceding steps. The allocation relies on the coupling (field accesses) between fields and classes.
- The final groups are suggested as new classes to replace the original God class.

The key steps are explained in detail in the following sections.

4.2 Computing Function-based Similarity

The function-based similarity (FS) measures the similarity between two methods concerning their functions. It is likely that two methods implementing similar functionalities had better been allocated to the same class during God class decomposition. However, the computation of function-based similarity could be challenging [11]. To this end, in this paper, we leverage the language model Sentence-T5 [35] to compute the function-based similarity.

ST5 is a fine-tuned version of SBERT [40], specially trained with a public dataset (i.e., CodeSearchNet [26]) by mapping source code to their functional descriptions. It could turn a piece of source code into a digital vector, and the resulting vector may represents its functional information and could be employed for various software engineering tasks[13]. In this paper, we leverage Sentence-T5 to embed each method into a 768-dimensional dense vector. After that, we compute the function-based similarity(FS) between the two methods by first computing the *Cosine* similarity of their corresponding vectors:

$$FS'(m_i, m_j) = \frac{\overrightarrow{m_i} \cdot \overrightarrow{m_j}}{\left\|\overrightarrow{m_i}\right\| \cdot \left\|\overrightarrow{m_j}\right\|} \quad (1)$$

where $\overrightarrow{m}$ is the vector of method m. Notably, the vectors generated by ST5 may contain negative numbers, and thus $FS'(m_i, m_j)$ could be negative as well. To ensure that the similarity value falls between 0 and 1, we perform linear normalization on FS':

$$FS(m_i, m_j) = \frac{FS'(m_i, m_j) - minFS'}{maxFS' - minFS'} \tag{2}$$

where $minFS'$ and $maxFS'$ represent the minimal and maximal values of FS' for a given God class.

4.3 Computing Position-based Similarity

According to our experience, physically adjacent methods are often closely related in semantics as well, and thus in most cases, they should be allocated to the same classes during God class decomposition.

To measure how close two elements are, we compute the position-based similarity (PS) between two methods m_i and m_j as follows:

$$PS(m_i, m_j) = \frac{1}{|p_{m_i} - p_{m_j}|} \tag{3}$$

where p_m is the relative position (index) of the method m. If its enclosing class has declared i elements before m, p_m equals $i + 1$. Notably, PS ranges from zero to one. If two methods are adjacent, their position-based similarity is maximized (one).

4.4 Computing Metrics-based Similarity

Code metrics, especially coupling and cohesion between software entities, have been widely exploited for the automated split of large software entities [25] [2]. Consequently, *ClassSplitter* employs code metrics as well for God class decomposition. More specifically, it leverages two metrics, SimD [25] and call-based dependence between methods (CDM) [7].

SimD was proposed by Gui and Scott [25] to measure the cohesion between two methods. Suppose that instance variables accessed by methods m_i and m_j are represented as two sets V_{m_i} and Vm_j, respectively, $SimD(m_i, m_j)$ is computed as follows:

$$SimD(m_i, m_j) = \frac{|V_{m_i} \cap V_{m_j}|}{|V_{m_i} \cup V_{m_j}|} \tag{4}$$

where $V_{m_i} \cap V_{m_j}$ is the overlap between two sets V_{m_i} and Vm_j, representing the instance variables accessed by both m_i and m_j. Consequently, $SimD(m_i, m_j)$ measures to what extent V_{m_i} and Vm_j overlap. $SimD$ ranges between zero and one, suitable to be taken as a similarity. Notably, instance variables accessed by a method refer to all variables (fields) defined outside the method but accessed by the method.

The call-based dependence between methods (CDM) was proposed by Bavota et al. [7] to measure the coupling between two methods concerning their inter-invocations. Let $Calls(m_i, m_j)$ be the number of calls performed by method m_i to method m_j, and $Calls_{in}(m_j)$ be the total number of incoming calls to m_j, the call-based dependence from m_i to m_j is computed as follows:

$$DirectedCDM(m_i, m_j) = \begin{cases} \frac{Calls(m_i, m_j)}{Calls_{in}(m_j)} & \text{if} \quad Calls_{in}(m_j) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

$DirectedCDM(m_i, m_j)$ represents how many percentages of the invocations of m_j are made by m_i. Notably, $DirectedCDM(m_i, m_j)$ has directions, and thus $DirectedCDM(m_i, m_j)$ is not necessarily equivalent to $DirectedCDM(m_j, m_i)$. To make the metrics commutative, Bavota et al. [7] defined the final commutative coupling metrics as follows:

$$CDM(m_i, m_j) = \max\left(DirectedCDM(m_i, m_j), DirectedCDM(m_j, m_i)\right) \tag{6}$$

We notice that $SimD(m_i, m_j)$ and $CDM(m_i, m_j)$ measure the coupling between two methods (m_i to m_j) concerning different aspects, i.e., variable access and method invocation. Consequently, in this paper, we leverage both of them as follows:

$$MS(m_i, m_j) = \frac{SimD(m_i, m_j) + CDM(m_i, m_j)}{2} \tag{7}$$

$MS(m_i, m_j)$ is the metrics-based similarity between methods m_i and m_j. It equals the average of $SimD(m_i, m_j)$ and $CDM(m_i, m_j)$. We take the average instead of the summary because the former ranges from zero to one whereas the latter could be larger than one.

Algorithm 1: Function-based Clustering

```
Input: $FS$ //Function-based similarity
    $methods$ //all methods within the God class
Output: $methodGroups$ // Initial groups of methods
1  // represent methods as a graph
2  $Graph$=CreateGraph($methods$);
3  // get the edges sorted by FS value
4  $edges \leftarrow \emptyset$;
5  size=$Graph$.NumberOfNodes();
6  for $i=0;i<size;i++$ do
7      for $j=i+1;j<size;j++$ do
8          $edges$.append($methods$[i], $methods$[j], $FS$[i][j]);
9      end
10 end
11 // Sort edges in descending order
12 $edges$.sortByWeight();
13 // connect methods with high similarity
14 for $k=0; k<edges.size*\theta; k++$ do
15     $Graph$.addEdge($edges$[k]);
16     if $Graph.IsConnected()$ then
17         $Graph$.removeEdge($edges$[k]);
18         break;
19     end
20 end
21 // represent the graph as a few groups
22 $methodGroups \leftarrow \emptyset$;
23 $subGraphs$=$Graph$.getConnectedSubgraphs();
24 foreach $g$ in $subGraphs$ do
25     $group$ = g.toList();
26     $methodGroups$.append($group$);
27 end
28 return $methodGroups$
```

4.5 Clustering-based Decomposition

In the preceding sections, we have computed a set of similarities between methods. Based on these similarities, in this section, we propose a clustering-based approach to decompose a given God class. Overall, the approach is composed of four parts. In the first part, it clusters methods according to function similarity. In the second part, it adapts the initial clusters by splitting inconsecutive clusters (that are physically split by methods outside the clusters) to ensure that all resulting clusters are consecutive blocks. It also merges all adjacent small clusters. In the third part, it merges small clusters. Finally, it allocates fields to clusters (of methods), and the resulting clusters are recommended as new (and smaller) classes to replace the God class. All four parts of the clustering algorithm are explained in detail in the following paragraphs.

4.5.1 Function-based Clustering. The first part of the clustering called function-based clustering, is presented in Algorithm 1. The algorithm takes the methods within the God class as well as their function-based similarity (*FS*) as input. It should decompose the methods into several groups (clusters) noted as *methodGroups*.

In the first place, it represents all methods as a graph (Line 2 of Algorithm 1). That is, it creates a unique node for each of the methods, but no edges are added. Lines 4-10 collect potential edges between methods with their function-based similarity as the weight of the edges. The edges are sorted in descending order (Line 12). Lines 14-20 try to connect methods with high similarity by adding the edges to the graph (Line 15). The algorithm would add the top θ percentages (empirically set to 8% with a small set of samples) of the edges unless the graph becomes a *connected graph*, i.e., all methods are connected. In case the newly added edge makes the graph a connected graph (Line 16), the algorithm would remove the edge so that the graph is not connected. Lines 22-27 represent each connected subgraph as a group of methods. All such groups are added to a list *methodGroups* that is finally returned by the algorithm.

4.5.2 Position-based Clustering. The second part of the clustering, called position-based clustering, is presented in Algorithm 2. The algorithm takes the groups of methods (*methodGroups*) generated by the preceding section (i.e., function-based clustering) as well as the methods' position-based similarity (*PS*) as input. It adapts the groups in *methodGroups* and returns the modified *methodGroups* as output.

The algorithm takes two steps to modify *methodGroups*. The first step, as shown in Lines 2-12, is to split inconsecutive clusters into smaller but consecutive clusters. A cluster (of methods) is consecutive if and only if

- It is composed of a single method or
- For any pair of methods (m_1 and m_2) within the cluster, there does not exist any method that is 1) not belonging to the cluster and 2) physically located between m_1 and m_2.

More vividly, a cluster of methods is consecutive if all such methods form a consecutive block within the God class. If it is not consecutive, we spit it into several consecutive ones. As a result of the splitting, all clusters are consecutive.

Algorithm 2: Position-based Clustering

```
   Input: PS //Position-based similarity
   methodGroups // Initial clusters of methods in the God class
   Output: methodGroups // Adapted clusters of methods
 1 //splitting inconsecutive clusters
 2 tempMethodGroups ← ∅;
 3 size=methodGroups.size();
 4 for i=0;i<size;i++ do
 5     if isInconsecutive(methodGroups[i], PS) then
 6         groups = methodGroups[i].splitByPosition(PS);
 7         tempMethodGroups.appendAll(groups);
 8     else
 9         tempMethodGroups.append(methodGroups[i]);
10     end
11 end
12 methodGroups ← tempMethodGroups;
13 tempMethodGroups ← ∅;
14 //merging successive singleton groups
15 methodGroups.sortByPosition();
16 size=methodGroups.size();
17 for i=0;i<size;i++ do
18     if !IsSingletonGroup(methodGroups[i]) then
19         tempMethodGroups.add(methodGroups[i]);
20         continue;
21     else
22         //Looking for successive singleton groups
23         for j=i+1;j<size;j++ do
24             if IsSingletonGroup(methodGroups[j]) then
25                 SingletonGroups.add(methodGroups[j]);
26             else
27                 break;
28             end
29         end
30         if SingletonGroups.size() > 1 then
31             //merge successive singleton groups
32             g = MergeGroups(SingletonGroups);
33             tempMethodGroups.add(g);
34             // skip all merged singleton groups
35             i=i+SingletonGroups.size() − 1;
36         else
37             tempMethodGroups.add(methodGroups[i]);
38         end
39     end
40 end
41 methodGroups ← tempMethodGroups;
```

In the second step, the algorithm merges some small and successive clusters that are composed of a single method. We call such groups *singleton groups.* The algorithm ranks the clusters by position (Line 15), enumerates all groups (Lines 16-40), and validates whether a given group is a singleton (Line 18). If yes (the ELSE branch begins at Line 21), it validates whether the following clusters are singleton groups as well. If it is followed by one or more singleton groups, all such groups are merged with it (Lines 30-33).

Algorithm 3: Similarity-based Merging

```
   Input: MS, PS, FS, //Similarity metrics
   methodGroups // Initial groups of methods
   Output: methodGroups //Final groups of methods
1  // keep merging the most similar groups
2  unmergeablePairs=∅;
3  while methodGroups.size() > 2 do
4      //retrieve two groups with the greatest similarity,
         excluding unmergeable pairs
5      SimGroups=
         FindMostSimilarGroups(methodGroups,unmergeablePairs);

6      if SimGroups==∅ then
7          break;
8      end
9      // try to merge the groups
10     newGroup=Merge(SimGroups);
11     if TooBig(newGroup) and NotTooClose(newGroup) then
12         // stop merging if the new group is too big
13         unmergeablePairs.add(newGroup);
14     else
15         //merge and replace
16         methodGroups.remove(SimGroups);
17         methodGroups.add(newGroup);
18     end
19 end
20 return methodGroups
```

Otherwise, no merging is conducted (Line 37). Notable, if multiple groups are merged, we increase the iteration index i by the number of merged clusters so that the merged clusters will not be enumerated again by the *FOR* iteration on Line 17.

4.5.3 Similarity-based Merging. The third part of the clustering called similarity-based merging, is presented in Algorithm 3. The algorithm takes the initial groups of methods (*methodGroups*) generated by the preceding sections as well as three categories of similarities as input. The major operation of the algorithm is to merge groups according to similarities.

The key of the algorithm is a loop (Lines 3-19). On each iteration of the loop, the algorithm first retrieves two groups with the highest similarity (Line 5) and tries to merge them (Line 10). If the resulting group is too big (Line 11), the algorithm would discard merging and notate the pair of groups as unmergeable pairs (Line 13). As a result, on the next iteration, the method invocation *FindMostSimilarGroups* on Line 5 would ignore this pair. The method *FindMostSimilarGroups* (Line 5 of Algorithm 3) returns a pair of highly similar groups. The overall similarity between two groups g_1 and g_2 is defined as follows:

$$OverallSim(g_1, g_2) = \frac{\#Hsm_{MS}(g_1, g_2) + \#Hsm_{PS}(g_1, g_2) + \#Hsm_{FS}(g_1, g_2)}{|g_1| * |g_2|} \tag{8}$$

where $|g_i|$ the number of methods within group g_i. $\#Hsm_{MS}(g_1, g_2)$, $\#Hsm_{PS}(g_1, g_2)$ and $\#Hsm_{FS}(g_1, g_2)$ represent the number of highly similar method pairs $< m_1 \in g_1, m_2 \in g_2 >$ concerning the metrics-based similarity (as defined in Section 4.4), position-based similarity(as defined in Section 4.3), and function-based similarity (as defined in Section 4.2), respectively. Two methods have high metrics-based similarity if their similarity is greater than 95% of the other metrics-based similarity; Two methods have high position-based similarity if they are separated by no more than three methods; Two methods have high function-based similarity if their similarity is greater than 93% of the other metrics-based similarity. Note that all such thresholds are set empirically with a small number of samples. They could be reset if needed.

We notice that if the new group is too big and the relationship is not very close(Line 11), the algorithm would give up merging. The new group merged from groups g_1 and g_2 is too big for merging if and only if

- The group consists of more than 85% of the source code in the God class, and
- The group contains more than 85% of the methods within the God class, and
- The similarity between g_1 and g_2 is smaller than 0.05.

The threshold (i.e., "2") on Line 3 is employed to guarantee that the algorithm would not merge all groups into a single group. Otherwise, the proposed approach cannot suggest any splitting solutions. Notably, since the iteration could be broken (Line 7), the number of the resulting groups is not necessarily two. This allows our approach to provide reasonable quantities and sizes of new classes for both complex and simpler God class inputs.

4.5.4 Access-based Field Allocation. In the preceding sections, we have divided the methods into a few number of groups. Each of the groups should represent a new class. However, the fields within the God class have not yet been allocated to the new classes. To this end, in this section, we allocate them into the new classes according to code coupling and element positions.

In the first step, we try to assign a field to the class that accesses the field more frequently than any of the other classes. The frequency of field access is counted by static source analysis. That is, every field access statement is counted equally as one time of access, no matter how many times the statement would be involved. For the remaining fields that have not been allocated for various reasons, e.g., not accessed by any classes, we allocate them according to their neighboring fields. For a given field f to be allocated and a candidate class c, the distance-based similarity between f and c is computed as follows:

$$pSim(f, c) = \sum_{fd \in c} \frac{1}{|p(f) - p(fd)|} \tag{9}$$

where $p(f)$ and $p(fd)$ represent the positions of fields f and fd (within the original God class), respectively. If c does not contain any field yet, the similarity is zero. Field f is allocated to the class with the greatest distance-based similarity to it. In case of a tie, we randomly select one of the winners.

5 EVALUATION

5.1 Research Questions

- RQ1: Can *ClassSplitter* improve the state of the art in the decomposition of God classes?
- RQ2: How do different types of similarities (i.e., position-based similarity, function-based similarity, and metrics-based similarity) employed by ClassSplitter influence its overall performance?
- RQ3: To what extent will the performance of ClassSplitter decrease if we replace the complex clustering in Section 4 with a traditional distance-based clustering algorithm?

ClassSplitter is built on the assumption that physically adjacent software entities have a greater chance to be allocated into the same classes during God class decomposition. RQ1 concerns the performance of ClassSplitter with regard to the state-of-the-art approaches. To answer RQ1, we compared ClassSplitter against the approach proposed by Bavota et al. [7], the approach proposed by Akash et al. [1], and the approach proposed by Alzahrani and Musaad [2]. Note that the papers did not explicitly name the approaches, and thus for convenience in this paper, we call them Bavota's, Akash's, and Alzahrani's, respectively. They were selected for comparison because they were the latest ones (in this line) that we can find, and thus they represent the state of the art for God class decomposition. RQ2 concerns the impact of the three types of similarities exploited by ClassSplitter, i.e., position-based similarity, function-based similarity, and metrics-based similarity. Answering RQ2 would validate the usefulness of the exploited similarities. RQ3 concerns the usefulness of the complex clustering algorithm proposed in Section 4. To answer this question, we should replace it with a simple distance-based clustering algorithm, and investigate how the performance of ClassSplitter changes. Answering this question helps reveal the necessity of designing a novel clustering algorithm for ClassSplitter.

5.2 Dataset

To evaluate ClassSplitter and the baseline approaches, we constructed a testing dataset by collecting (from Github) real-world God classes and their solutions that had been proposed and conducted by the original developers. The data collection followed exactly the same methodology once employed to construct the case study dataset in Section 3.2. Note that we collected data from top 1000 Java projects in Github except for those that had been involved in the case study in Section 3.2. The exclusion guranatees that the testing dataset does not overlap with the case study dataset.

We finally collected 133 God classes and their original (reference) solutions for evaluation. Each of the solutions is represented as a set of classes that had been used to replace the corresponding God class. Such solutions would be employed as a Golden set to quantitatively assess the solutions proposed by the evaluated approaches, i.e., to what extent the proposed solutions are similar to the original ones.

5.3 Performance Metrics

To quantitatively assess the performance of the evaluated approaches, including both ClassSplitter and the baselines, we reuse the performance metrics proposed by Wen and Tzerpos [47]. The metrics, called *MoJoFM*, is computed as follows:

$$MoJoFM(S, REF) = 100\% - \frac{mno(S, REF)}{max(mno(\forall S, REF))} \tag{10}$$

where S is the solution to be evaluated and REF is the reference solution retrieved from the golden set. $mno(S, REF)$ is the minimum number of move or join operations to transform a partition (solution) S to another partition REF. It is also called *distance* between S and REF. $max(mno(\forall S, REF))$ is the maximum possible distance from REF to any possible partition (solution) for the same God class. Notably, MoJoFM was designed to evaluate a single solution for a single God class. To compare various solutions proposed by the evaluated approaches for various God classes, we calculated the average MoJoFM for each of the evaluated approaches by averaging the MoJoFM on all solutions generated by a single approach. Cohesion and coupling code metrics are not used for quantitative performance evaluation as they have already been utilized by the proposed approach and baselines to guide the decomposition of complex classes. Assessing their impact with the same metrics could potentially overstate their performance.

Besides the quantitative metrics, i.e., MoJoFM, we also measured the quality of suggested solutions manually and qualitatively. The manual qualitative measurement followed the widely-used 5-point Likert Scale [18]. Consequently, for any recommended solution for a God class, the participants should rate its quality as one of the following five scales:

- Excellent (4 points), i.e., better than my solution;
- Good (3 points), i.e., comparable to my solution;
- Acceptable (2 points), i.e., reasonable but request small modification;
- Fair (1 point), i.e., deserving significant modification;
- Poor (0 points), i.e., unreasonable and useless.

We also provided a simplified rating by asking the participants to select the preferred solution for each of the God classes. That is, FOR each God class, we presented the solution recommended by our approach and the solutions suggested by the baselines, and the participants should select the winner. Based on the selection, we computed how often ClassSplitter outperformed the baseline. To reduce the potential bias, we employed an anonymous review process where the participants did not know the recommender (ClassSplitter or the baselines) of the solutions. We requested three participants to rank all 133 items and thus assessed the consistency among them. All of the participants had more than three years of Java programming experience. Notably, since we had four evaluated approaches and the manual ranking could be tedious and time-consuming, we only selected the top 2 of the evaluated approaches (ranked by quantitative metrics, i.e., their average MoJoFM) for the manual ranking.

5.4 Improving the State of the Art

To answer research question RQ1, we first compared ClassSplitter against three baselines, i.e., Bavota's, Akash's, and Alzahrani's. Notably, our approach and the first two baselines have several configuration parameters. To optimize their settings, we randomly sampled five God classes and searched for the best settings (regarding MoJoFM). The resulting settings ($w_{SSM} = 0.2$, $w_{CDM} = 0.2$, $w_{CSM} = 0.6$, $minCoupling = Q_3$ for Bavota's; $w_{SSM} = 0.2$,

w_{CDM} = 0.3, w_{CSM} = 0.5, *minCoupling* = 0.36 for Akash's; and θ = 0.08 for ClassSplitter) were used in the remaining cases. The evaluation results of average MoJoFM value are presented in Table 2. MoJoFM-M and MoJoFM-F are average MoJoFM of methods and fields. Note that *Alzahrani's* clusters methods only and fields are ignored. Consequently, its performance metrics on fields are unavailable.

From Table 2, we make the following observations:

- Bavota's outperformed the other two baselines. Its average MoJoFM in method clustering is 32.6%, substantially higher than that of Akash's (26.4%) and Alzahrani's (22.5%). Its performance (43.9%) in field clustering is also better than that of Akash's (30.2%).
- ClassSplitter resulted in substantially greater MoJoFM than the baselines. Compared to the best baseline (i.e., Bavaota's), it improved the average MoJoFM by 47%=(49.6%-33.6%)/33.6%, while the low p-value(0.00036) in the U test also suggests the significantly improvement in MoJoFM. The improvement in method clustering (61%=(52.6%-32.6%)/32.6%) is even larger.

According to the specification in Section 5.3, we requested developers to manually rank the best two approaches, i.e., ClassSplitter and Bavota's, and the reference solutions suggested by the original developers of the involved applications.. The results are presented in Table 3 where the fourth row specifies how often ClassSplitter's solutions were preferred (both absolute frequency and probability). From this table, we make the following observations:

- Solutions suggested by ClassSplitter were often preferred by developers. On 78% of the cases, its solutions were preferred whereas Vavota's solutions were preferred on only 7% (=1-78%-15%) of cases. In other cases (15%), they drew.
- The solutions suggested by ClassSplitter are considerably good. The average ranking is 2.21, between *good* (i.e., *comparable to my solution*, 3 points) and *acceptable* (i.e., *reasonable but request small modification*, 2 points). In contrast, the average ranking of Bavota's solutions is 0.92, lower than *fair* (i.e., *deserving significant modification*, 1 point).
- The ranking of reference solution is 3.15(between Good and Excellent), significantly higher than the solutions given by automatic refactoring tools. The results may suggest that the scores are positively related to the quality of the solutions.

Based on both quantitative (and objective) assessment and qualitative (and subjective) assessment, we conclude that ClassSplitter substantially improves the state of the art in God class decomposition. From a code metric perspective, ClassSplitter also significantly improves code quality: cohesion optimized from average 822 to 122 in LCOM metrics(smaller is better), with average coupling slightly increased 12%(251 to 282) in MPC metrics(smaller is better). ClassSplitter was efficient, taking 10.2 seconds on average to decompose a single God class.

5.5 Effect of Different Similarities

To validate the usefulness of the three categories of similarities, i.e., position-based similarity, function-based similarity, and metrics-based similarity, we disabled one of them at a time and repeated the evaluation. By comparing the overall performance of ClassSplitter with and without the given similarity, we may quantitatively reveal the impact of the given similarity. The evaluation results are presented in Table 4. While the second row of the table presents the performance of ClassSplitter in the default setting (i.e., all three categories of similarities were exploited), the following three rows present the performance of ClassSplitter when one of the three types of similarities was disabled.

Table 2: Improving the State of the Art

	ClassSplitter	Bavota's	Akash's	Alzahrani's
MoJoFM-M	52.6%	32.6%	26.4%	22.5%
MoJoFM-F	44.9%	43.9%	30.2%	–
MoJoFM	49.6%	33.6%	32.3%	–

Table 3: Participants' Ranking

Participant	A	B	C	Average
Reference Solution	3.13	3.19	3.14	3.15
ClassSplitter	2.21	2.04	2.38	2.21
Bavota's	0.88	0.83	1.05	0.92
#Wins	104(78%)	100(75%)	109(82%)	104(78%)
#Ties	26(20%)	24(18%)	8(6%)	19(15%)

Table 4: Effect of Different Similarities

Setting	Average MoJoFM		
	Methods & Fields	Methods	Fields
Enabling All	49.6%	52.6%	44.9%
Disabling Position	35.0%	30.1%	37.0%
Disabling Function	46.7%	48.2%	41.2%
Disabling Metrics	49.2%	51.5%	46.8%

From the table, we make the following observations:

- First, all of the employed similarities are useful. Disabling any of the similarities resulted in a reduction in performance.
- Second, position-based similarity had the greatest impact on the overall performance of ClassSplitter. Disabling it resulted in the greatest reduction in performance: The reduction on average method MoJoFM was 22.5 percentage points (pp) =52.6%-30.1% whereas the reduction caused by disabling other similarities varied from 4.4 pp=52.6%-48.2% (disabling function-based similarity) and 1.1 pp=52.6%-51.5% (disabling metrics-based similarity).
- Third, exploiting similarities had a greater impact on the clustering of methods than the clustering of fields. For example, by exploiting the position-based similarity, ClassSplitter improved the performance in method clustering by 75%=(52.6%-30.1%)/30.1% whereas the performance in field clustering was improved by 21%=(44.9%-37.0%)/37.0%. The major reason for the difference is that most fields are allocated according to field accesses instead of position-based similarity. However, exploiting position-based similarity would influence the clustering of methods, and the latter in turn would influence the clustering (allocation) of fields. Consequently, exploiting such metrics would finally influence the clustering of fields.

We conclude based on the preceding analysis that all three categories of similarities are useful and the position-based similarity is by far the most valuable. The results also validate the usefulness of relative positions of methods/fields in God class decomposition.

5.6 Effect of Customized Clustering Algorithm

To improve the performance of ClassSplitter, we customized a multiple-step clustering algorithm in Section 4.5 to cluster methods according to the position-based similarity, function-based similarity, and metrics-based similarity between methods. To validate the necessity of the customized clustering algorithm as specified in Section 4.5, we replaced it with a traditional distance-based clustering algorithm where distances are computed as follows :

$$distance(m_1, m_2) = -(MS(m_1, m_2) + FS(m_1, m_2) + PS(m_1, m_2)) \quad (11)$$

$MS(m_1, m_2)$, $FS(m_1, m_2)$, and $PS(m_1, m_2)$ represent the metrics-based similarity, function-based similarity, and position-based similarity between methods m_1 and m_2, respectively. Consequently, the more similar the two methods are, the smaller their distance is. Notably, the clustering algorithm in Section 4.5 is used to cluster methods only, and thus we did not evaluate how it may influence the allocation of fields.

Our evaluation results suggest that replacing the customized clustering algorithm would substantially reduce the performance of method clustering. The average MoJoFM was reduced from 52.6% to 44%. In other words, customizing the clustering algorithm improved the performance by 20%=(52.6%-44%)/44%. We conclude based on the analysis that customizing the clustering algorithm is critical for the performance of ClassSplitter.

5.7 Threats to Validity

A threat to internal validity is that the conclusions were drawn on a limited number of cases (133 God classes in the evaluation and 54 classes in the empirical study). Special characters of such cases may bias the conclusions. Noting that the data collection is tedious and time-consuming, involving extensive human intervention. Consequently, it is difficult to significantly enlarge the datasets. We collected data from projects developed by different teams/companies, and come from different domains, e.g., Android framework, Android Apps, distributed computing, data engineering, software testing, decompiler, Android application development environment, cloud-native microservices, database, communication, and game engines. Such projects have different numbers of developers (varying from 1 to 1480 with a median of 26), different numbers of commits (varying from 5 to 34186, with a median of 3954), and different histories (varying from 2 days to 14 years, with a median of 3 years). The size of the God classes varies from 64 to 4869 LOC (with a median of 470 LOC), and the number of methods/fields within a single God class varies from 20 to 270 (with a median of 43). All such diverse statistics may suggest that the datasets have a good diversity, and may reduce the threat from the size of the dataset.

The first threat to construct validity is that the benchmark employed by the evaluation could be inaccurate. While computing the performance metrics (i.e., MoJoFM), we compared the suggested solutions against solutions in the benchmark (i.e., actual solutions proposed and conducted by the original developers). However, solutions different from the golden set could be reasonable (and correct) as well because the decomposition of classes is often subjective and there could be multiple good solutions for a single God class. To reduce the threat, we also requested experienced developers to rank the suggested solutions, and the manual ranking confirmed the conclusions drawn with objective performance metrics. However, the manual ranking could be inaccurate as well. To reduce the threat, we requested experienced developers with at least three years of Java experience. We also requested multiple participants to rank each of the cases, and the resulting Kappa coefficient (0.57) suggested a moderate agreement between participants.

The second threat to construct validity is that the data collection in Section 3 and Section 5 is subjective and thus the resulting God classes could be debatable. To reduce the threat, we requested multiple experienced participants (with more than 5 years of Java programming experiences) to collect data independently, and a Kappa coefficient (0.492) suggested a moderate agreement between participants. We also requested them to discussed together concerning the inconsistent cases, and they managed to reach a consensus on all cases. We also publish the dataset [15] for further validation.

The third threat to construct validity is that all the data collected in this research are from the most stared java project on GitHub. This would result in the most high-quality projects, which in turn may lead to bias. Notably, selecting projects by stars is one of the most common strategies for selecting projects in papers related to software refactoring. The most recent example is the paper by Dong et al. [21]. Besides, refactoring solutions discovered from such high-quality projects are often of high quality, which helps reduce the threats to construct validity.

6 CONCLUSIONS AND FUTURE WORK

Decomposing God classes is challenging, and thus automated decomposition is highly preferred. To this end, in this paper, we propose a new approach to decomposing God classes. The approach is novel in that it exploits features (like positions of methods and fields) that have not yet been exploited by existing approaches in this line, and it customizes a clustering algorithm, especially for God class decomposition. The key insight has been validated by a study presented in the paper, and the effect of the proposed approach has been validated with real-world God classes. Our evaluation results suggest that the proposed approach can substantially improve the state of the art in God class decomposition.

Although the proposed approach is not language-specific, the prototype implementation accepts Java programs only. The code analysis component (that extracts code metrics from source code) is language-specific, and it should be re-implemented if additional programming languages should be considered in the future. Considering the similarity between *God class* and *long method*, it is potentially fruitful in the future to decompose long methods with the key inside presented in this paper.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers from ASE 2024 for their insightful comments and constructive suggestions. This work was partially supported by the National Natural Science Foundation of China (62232003 and 62172037), China National Postdoctoral Program for Innovative Talents (BX20240008) and CCF-Huawei Populus Grove Fund (CCF-HuaweiSE202411).

REFERENCES

[1] Pritom Saha Akash., Ali Zafar Sadiq., and Ahmedul Kabir. 2019. An Approach of Extracting God Class Exploiting Both Structural and Semantic Similarity. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*. INSTICC, SciTePress, 427–433. https://doi.org/10.5220/0007743804270433

[2] Musaad Alzahrani. 2022. Extract Class Refactoring Based on Cohesion and Coupling: A Greedy Approach. *Computers* 11, 8 (2022), 123.

[3] Nicolas Anquetil, Anne Etien, Gaelle Andreo, and Stéphane Ducasse. 2019. Decomposing god classes at siemens. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 169–180.

[4] Nicolas Anquetil and Timothy C Lethbridge. 1999. Experiments with Clustering as A Software Remodularization Method. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*. IEEE, 235–255.

[5] ant media. 2017. Ant-Media-server. https://github.com/ant-media/Ant-Media-Server

[6] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and Experimenting Machine Learning Tchniques for Code Smell Detection. *Empirical Software Engineering* 21 (2016), 1143–1191.

[7] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. Automating Extract Class Refactoring: An Improved Method and its Evaluation. *Empirical Software Engineering* 19 (2014), 1617–1664.

[8] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. 2011. Identifying Extract Class Refactoring Opportunities using Structural and Semantic Cohesion Measures. *Journal of Systems and Software* 84, 3 (2011), 397–414.

[9] Kent Beck, Martin Fowler, and Grandma Beck. 1999. Bad Smells in Code. *Refactoring: Improving the design of existing code* 1, 1999 (1999), 75–88.

[10] Neeraj Bhargava, Girja Sharma, Ritu Bhargava, and Manish Mathuria. 2013. Decision Tree Analysis on j48 Algorithm for Data Mining. *Proceedings of international journal of advanced research in computer science and software engineering* 3, 6 (2013).

[11] David Binkley. 2007. Source Code Analysis: A Road Map. In *Future of Software Engineering (FOSE '07)*. 104–119. https://doi.org/10.1109/FOSE.2007.27

[12] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent Dirichlet Allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[13] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When Deep Learning Met Code Search *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 964–974. https://doi.org/10.1145/3338906.3340458

[14] Xiangping CHEN, Xing HU, Yuan HUANG, He JIANG, Weixing JI, Yanjie JIANG, Yanyan JIANG, Bo LIU, Hui LIU, Xiaochen LI, Xiaoli LIAN, Guozhu MENG, Xin PENG0, Hailong SUN, Lin SHI, Bo WANG, Chong WANG0, Jiayi WANG, Tiantian WANG, Jifeng XUAN, Xin XIA, Yibiao YANG, Yixin YANG, Li ZHANG, Yuming ZHOU, and ZHANG Lu. [n. d.]. Deep Learning-based Software Engineering: Progress, Challenges, and Opportunities. *SCIENCE CHINA Information Sciences* ([n. d.]), –. https://doi.org/10.1007/s11432-023-4127-5

[15] ClassSplitter. 2023. ClassSplitter. https://github.com/ClassSplitter/ClassSplitter

[16] William W. Cohen. 1995. Fast Effective Rule Induction. In *Machine Learning Proceedings 1995*, Armand Prieditis and Stuart Russell (Eds.). Morgan Kaufmann, San Francisco (CA), 115–123. https://doi.org/10.1016/B978-1-55860-377-6.50023-2

[17] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. MIT press.

[18] John Dawes. 2008. Do Data Characteristics Change According to the Number of Scale Points Used? An Experiment using 5-point, 7-point and 10-point Scales. *International journal of market research* 50, 1 (2008), 61–104.

[19] Andrea De Lucia, Rocco Oliveto, and Luigi Vorraro. 2008. Using Structural and Semantic Metrics to Improve Class Cohesion. In *2008 IEEE International Conference on Software Maintenance*. 27–36. https://doi.org/10.1109/ICSM.2008.4658051

[20] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. 1990. Indexing by Latent Semantic Analysis. *Journal of the American society for information science* 41, 6 (1990), 391–407.

[21] Chunhao Dong, Yanjie Jiang, Nan Niu, Yuxia Zhang, and Hui Liu. 2024. Context-Aware Name Recommendation for Field Renaming. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 235, 13 pages. https://doi.org/10.1145/3597503.3639195

[22] Marios Fokaefs, Nikolaos Tsantalis, Alexander Chatzigeorgiou, and Jorg Sander. 2009. Decomposing Object-Oriented Class Modules using An Agglomerative Clustering Technique. In *2009 IEEE International Conference on Software Maintenance*. IEEE, 93–101.

[23] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. 2011. Jdeodorant: Identification and Application of Extract Class Refactorings. In *Proceedings of the 33rd International Conference on Software Engineering*. 1037–1039.

[24] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code Smell Severity Classification using Machine Learning Techniques. *Knowledge-Based Systems* 128 (2017), 43–58.

[25] G. Gui and P. D. Scott. 2006. Coupling and Cohesion Measures for Evaluation of Component Reusability. In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (Shanghai, China) *(MSR '06)*. Association for Computing Machinery, New York, NY, USA, 18–21. https://doi.org/10.1145/1137983.1137989

[26] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet Challenge: Evaluating the state of Semantic Code Search. *arXiv preprint arXiv:1909.09436* (2019).

[27] Tahmim Jeba, Tarek Mahmuda, Pritom S Akashb, and Nadia Naharb. 2020. God Class Refactoring Recommendation and Extraction using Context Based Grouping. (2020).

[28] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2009. A Bayesian Approach for the Detection of Code and Ddesign Smells. In *2009 Ninth International Conference on Quality Software*. IEEE, 305–314.

[29] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2011. BDTEX: A GQM-based Bayesian Approach for the Detection of Antipatterns. *Journal of Systems and Software* 84, 4 (2011), 559–572.

[30] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2021. Deep Learning Based Code Smell Detection. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1811–1837. https://doi.org/10.1109/TSE.2019.2936376

[31] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, and Esma Aimeur. 2012. Smurf: A Svm-based Incremental Antipattern Detection Approach. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 466–475.

[32] Radu Marinescu. 2004. Detection Strategies: Metrics-based Rules for Detecting Design Flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings*. IEEE, 350–359.

[33] Robert Cecil Martin. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR.

[34] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Transactions on software engineering* 30, 2 (2004), 126–139.

[35] Jianmo Ni, Gustavo Hernandez Abrego, Noah Constant, Ji Ma, Keith B. Hall, Daniel Cer, and Yinfei Yang (Eds.). 2022. *Sentence-T5: Scaling up Sentence Encoder from Pre-trained Text-to-Text Transfer Transformer*. https://aclanthology.org/2022.findings-acl.146/

[36] William F Opdyke. 1992. *Refactoring Object-Oriented Frameworks*. University of Illinois at Urbana-Champaign.

[37] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2014. Mining Version Histories for Detecting Code Smells. *IEEE Transactions on Software Engineering* 41, 5 (2014), 462–489.

[38] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. 2016. A Textual-based Technique for Smell Detection. In *2016 IEEE 24th international conference on program comprehension (ICPC)*. IEEE, 1–10.

[39] José Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow. 2022. Code Smells Detection and Visualization: A Systematic Literature Review. *Archives of Computational Methods in Engineering* 29, 1 (2022), 47–94.

[40] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence Embeddings using Siamese Bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[41] İbrahim Şanlıalp, Muhammed Maruf Öztürk, and Tuncay Yiğit. 2022. Energy Efficiency Analysis of Code Refactoring Techniques for Green and Sustainable Software in Portable Devices. *Electronics* 11, 3 (2022), 442.

[42] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. 2021. Code Smell Detection by Deep Direct-learning and Transfer-learning. *Journal of Systems and Software* 176 (2021), 110936.

[43] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. 2001. Metrics based Refactoring. In *Proceedings fifth european conference on software maintenance and reengineering*. IEEE, 30–38.

[44] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367.

[45] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[46] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A Survey of Transfer Learning. *Journal of Big data* 3, 1 (2016), 1–40.

[47] Zhihua Wen and V. Tzerpos. 2004. An Effectiveness Measure for Software Clustering Algorithms. In *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004*. 194–203. https://doi.org/10.1109/WPC.2004.1311061