Received: Added at production | Revised: Added at production | Accepted: Added at production
DOI: xxx/xxxx

RESEARCH ARTICLE - EMPIRICAL

Anti-bloater class restructuring: an exploratory study

João Paulo L. Machado[1] | Elder V. P. Sobrinho[2] | Marcelo A. Maia[1]

[1]Faculty of Computing, Federal University of Uberlândia, Uberlândia, MG, 38400-902, Brazil
[2]Electrical Engineering Department, Federal University of Triângulo Mineiro, Uberaba, MG, Brazil

Correspondence
*Marcelo A. Maia. Email: marcelo.maia@ufu.br

Summary

Proper software modularization still poses challenges to developers. One of the symptoms of inappropriate modularization is the large size of object-oriented classes. In that case, a possible solution would be class restructuring with refactorings, such as, Extract Class, Extract Super-class or Move Method. However, class refactoring is challenging because of the possible side effects of improper changes. In this context, more effective decision support systems on which classes are worthwhile for restructuring to improve modularity are still lacking. This work focuses on exploring possible alternatives for supporting decision on class restructuring. A prospective study was performed on selected kinds of restructuring, aiming at determining what types of strategies are typically adopted to restructure bloated classes, and which classes developers decided to restructure. Then, we proposed and evaluated a predictive model for indicating which classes to restructure, aiming at delivering a restructuring guide on those classes. Finally, we conducted a qualitative study to evaluate the perception of developers on such guides based on predictions for real software. The results have shown situations in which the proposed predictions could help the restructuring process, but also elucidated possible improvements and limitations.

KEYWORDS:
bloaters, structure, refactoring, temporal analysis, predictive models

1 | INTRODUCTION

Software systems are constantly modified by developers. These changes are intended to introduce new features, new user requests, fix bugs or optimize modularity[1,2]. Developers are expected to know the system design in order to identify where and how to properly change the source code. Currently, the available tools to assist developers during the maintenance process do not provide sufficient insight about the details of software design, and as consequence, average developers, other than those original architects, may not be able to identify an adequate location to change the system. If a change is made in an inappropriate place, it may decay the original design, or even violate dependency relations between the components of the architecture, generating inconsistencies and poor modularity. To avoid design decay, preventive actions are recommended to be carried out during the software life cycle.

In object-oriented systems, a preventive action is software refactoring, which aims to improve the structure and modularity of the system without changing its observable behavior[3]. There are different refactoring strategies which have specific characteristics that can impact the quality of the system. For instance, *Extract Class* is a widely used refactoring type that allows a class to be split into other two, where classes resulting from division will have the same responsibilities of the original class[3]. Depending on how the relationship between classes is established, the *Extract Class* can be specialized as *Extract Super-Class*, in which the extracted class becomes the parent of the class with the remaining elements[3]. Moreover, if a suitable target class already exists, elements can be moved to that class with, for instance, *Move Method*. Those kinds of refactoring can be characterized as anti-bloater class restructurings, which can be applied during software evolution in classes suffering from *bloater* code[4,5].

Developers need information to support the identification of classes that are not only bloated, but also, are worthwhile to change[6]. So, in this study, we aim to explore an alternative that should provide information to support class restructuring during the life cycle of a system. The strategy of our study includes the investigation of the life cycle of selected software systems to find changes that strongly improved a structural metric of a class. Our hypothesis is that whenever developers strongly improves the structure of a class, the respective change is motivated by a previous critical structural condition of the respective class. Whenever we find just slight improvements in the history of a class, possibly its structure is not critical enough, or those improvements are just coincidental with the whole change.

In this work, first we conduct a temporal analysis to understand the historical evolution of a structural metric that could unveil when a class restructuring was worthwhile to be carried out, inducing a major improvement on the respective metric. We expect that learning the features of a class, at which a restructuring was considered appropriate, we could suggest similar restructuring points. The exploratory study based on the temporal analysis of structural changes is aimed at finding classes that have undergone sharp restructurings and the characteristics of those restructurings that can help us better understand how they have been carried out. We capture these restructurings between release changes, so they may include refactoring, but also can change the observable behavior of the system, so we decide to refer them to as restructuring and not as refactoring.

Moreover, the knowledge of which classes have undergone restructuring during their life cycles is not much useful per se, so we also aim to translate the findings of the temporal analysis into a more practical approach that could be able to predict when a class is suitable for restructuring based solely on its current snapshot of structural metrics without referring to its past evolution. For that goal, we construct a prediction model that given a snapshot of the structural metrics of a class, it recommends if it is worthwhile to restructure or not that class.

Finally, in order to evaluate how the proposed approach for recommending class restructuring are perceived by developers, we conduct a survey with actual developers of the subject systems to understand their perception and evaluate the recommended strategies of the proposed approach.

We expect that the above three studies help to answer the following motivating questions on which classes and how to restructure them: **Q1 (which):** Is there a typical situation for a class, in terms of its quality structural metrics, that developers can use to decide restructuring that class? **Q2 (how):** Is it possible to detect a more appropriate strategy for restructuring a class?

To sum up, the major contributions of this paper are:

1. better understanding on how anti-bloater restructurings can be carried out during the life cycle of software systems;
2. a novel approach to identify bloated class restructuring opportunities. This approach is novel, in the sense that it has been conceived upon a decay indicator, whose training was carried out with data obtained from the prospective analysis of changes of a structural metric for classes;
3. the evaluation of the positive and negative impacts of proposed restructuring strategy based on the qualitative analysis of the perception of actual developers on the provided recommendations.

The rest of this paper is organized as follows. In Section 2, a motivating example shows how the temporal analysis of a structural metric can be useful for understanding restructuring opportunities. Section 3, an exploratory study is conducted to identify classes worthwhile to restructure and different kinds of restructuring strategies found on longitudinal data collected from several software releases. Section 4 presents the predictive model used to identify the restructuring opportunities. Section 5 presents a survey conducted with real developers on the recommendations provided for their systems. Section 6 presents the lessons learned answering the motivating questions. Section 7 discusses some threats to validity. Section 8 presents related work. Finally, the conclusions are presented.

2 | MOTIVATING EXAMPLE

As a motivating example, a Lucene[1] case is presented. Lucene is a open source tool for text search developed in Java. We first aimed to find in the issue tracker, open issues related to refactoring activities. So, we used the issue tracker's internal search engine and searched for problems related to the *refactoring*. We selected, for a manual inspection, only issues of the type improvement and with status *open*, and started reading each issue to find any interesting point related to refactoring bloated classes. One particular issue, "Lucene-2026", has the following title: *"Refactoring of IndexWriter"*. The *IndexWriter* class is responsible for creating indexes that are used to search text. That issue has shown that *IndexWriter* is a bloated class that developers aimed at somehow refactoring it into more cohesive classes.

So, we manually inspected the snapshots of the class *IndexWriter* between different releases to understand the major structural differences in those snapshots. Actually, we could find out that between releases 3.6.2 and 4.0.0, the *IndexWriter* class had already undergone restructuring.

[1]https://lucene.apache.org/

Then, we decided to manually inspect the evolution of some structural *IndexWriter* metrics (LCOM - Lack of Cohesion of Methods, for cohesion, and CBO - Coupling Between Objects, for coupling) to identify if there was any connection between the evolution of the metrics and the moment when the class had undergone restructuring.

From the manual inspection of the evolution of structural metrics of this class, we particularly observed that the LCOM metric had an interesting behavior. LCOM was originally aimed at measuring the cohesion level of the class by counting methods that are not related to each other by sharing attributes[7]. Although LCOM has been criticized in the literature[8], we still decided to analyze it because we compare the value of the metric within the same class, i.e., we do not compare LCOM of different classes. Moreover, we are not interested in analyzing the cohesion of the class, but the structure that LCOM represents. High LCOM values would presumably indicate that a class should be split into two or more classes[9], because of possible mixed functionality[10].

Figure 1 shows the evolution of the LCOM metric for the *IndexWriter* class. We can observe a sharp and significant drop of LCOM in release 4.0.0.

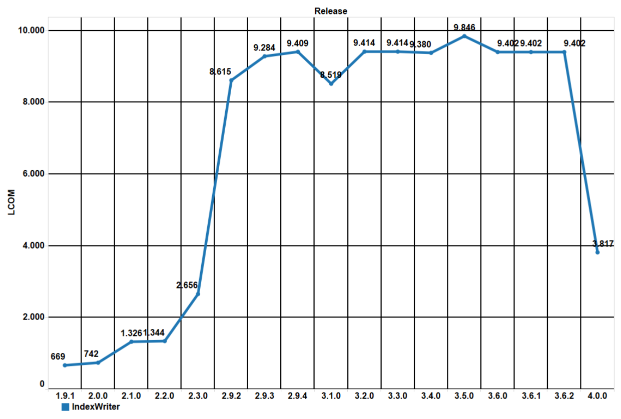

FIGURE 1 LCOM evolution of *IndexWriter*

We found out that this decrease between releases 3.6.2 and 4.0.0 was connected with the previously mentioned restructuring. We observed that several getter and setter methods that are responsible for reading and assigning values to the attributes of *IndexWriter* have been moved to a class called *IndexWriterConfig*. Listing 1 shows an example of a method that has been moved.

Listing 1: *Getter* method moved to *IndexWriterConfig*

```
/*Returns the current MergePolicy in use by this writer.
@see #setMergePolicy
@deprecated use {@link IndexWriterConfig#getMergePolicy()}*/
@deprecated
public MergePolicy getMergePolicy(){
        ensureOpen();
        return mergePolicy;
}
```

Interestingly, these methods had been marked as deprecated in earlier releases. A method is marked as deprecated when developers want to indicate that it will be removed later and/or that there is another method that can be used instead. Figure 2 presents an analysis of the evolution of LCOM in the *IndexWriterConfig* target class, which was created in the release where the *IndexWriter* methods were marked as deprecated. In this way, the *IndexWriterConfig* class was created with a *copy* of the getter and setter methods marked as deprecated in the *IndexWriter* class.

Moreover, we found out that since from release 3.1.0 some of those deprecated methods have been removed, and in 4.0.0 the remaining of the deprecated methods have been completely removed, completing a gradually performed Extract Class refactoring from the class *IndexWriter* to the class *IndexWriterConfig*. In the release 4.0.0, the methods *getter* and *setter* were available only in *IndexWriterConfig*.

This analysis showed that after completing the restructuring, the resulting classes have better cohesion than the original class. Therefore, an improvement in the modularization of the system has been observed.

So, in this work, we aim at investigating how the variation in the LCOM metrics of a class can indicate anti-bloater class restructuring, such as, Extract Class, Extract Superclass, Move Method (Section 4), and investigating the feasibility of a class restructuring predictor based on the structural metrics of only the current snapshot that could serve as a restructuring recommendation approach (Sections 5 and 6).

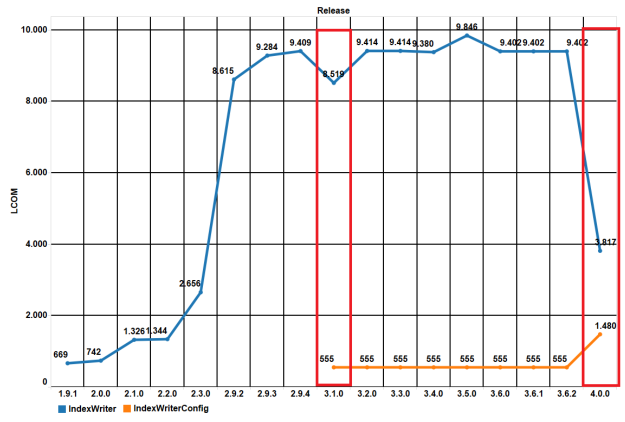

FIGURE 2 Gradual IndexWriter restructuring

3 | AN EXPLORATORY TEMPORAL STUDY OF CLASS STRUCTURE

In this section, we conduct a temporal analysis to capture structural changes in classes to identify those classes that had undergone important restructurings. The long-lived restructuring process conducted in the class *IndexWriter*, shown in Section 2, poses an important challenge in current state-of-the-art tools that mine refactoring from commits[11,12]. This kind of tool detects refactorings in specific commits, and thus they are not able to capture a restructuring that has been conducted within several commits that spanned possibly several releases. So, instead of mining refactoring commits in the version history, we opted to proceed with the analysis of *sharp* changes in a structural metric of a given class between two consecutive releases in order to possibly capture an important restructuring.

The main objective of this analysis is to determine the kinds of restructuring strategy was adopted when important structural changes take place in the life cycle of a class, and the list of classes that have undergone restructuring.

In this context, the source code of classes with significant changes related to class restructuring between two releases are manually compared. In order to capture these changes, we rely on the LCOM metric because we want to capture the movement of methods and their respective attributes to other possibly new classes, which indeed corresponds to the intent of a Extract Class / Move Method refactoring. When a class is split, LCOM tends to decrease because the number of methods that do not share attributes also tends to decrease. The analysis of the difference in LCOM of a class between two consecutive realease aims to determine if there is a significant number of methods that have been moved from this class to another class. In this case, an anti-bloater class restructuring could have occurred, and deserves a manual inspection to verify if that condition actually occurred.

During the manual inspection, we evaluate several points, such as, 1) the characteristics of the method being moved, if they a simple or complex; 2) the characteristics of the target class, if they are brand new or pre-existent; 3) the type of coupling between that target and source classes, if this coupling is based on an association or inheritance relationship; 4) what is the time frame in terms of releases between the creation of the target class and the structural change in the source class; 5) how much the target and source classes change their structural properties.

3.1 | Subject systems

To collect structural metrics during the life cycle of a system, we relied on the Apache ecosystem because releases are well-characterized and we could conveniently capture the code snapshots for them. In that ecosystem, we selected eight Java systems. We used the tool CKJM *extended* to collect structural and quality metrics for all classes in all available releases. Table 1 summarizes the number of releases and the number of classes considered in all respective releases. A class typically occur in more than one release, but not necessarily on all of them.

3.2 | Class selection procedure

To choose the classes for manual inspection to define if they have undergone a non-negligible restructuring process, we define a strategy to help finding those classes more easily. The class selection strategy is based on the change of the LCOM metric of that class from one release to another. After classes have been selected, we manually inspected the source code of those classes between the respective releases to find out, which kind of refactoring (Extract (Super-)Class or Move Method), if any, has been carried out. This manual procedure was conducted with an inspection

System	# Releases	# Classes
JMeter	36	1090
Ant	30	1558
Javacc	12	180
Jetty	167	2430
Log4j	48	1767
Lucene	58	3576
Tomcat	207	4556
Xerces	36	1921

TABLE 1 Subject Systems: releases and classes

TABLE 2 Computed ΔLCOM's

Δ LCOM's	#	%
$\Delta LCOM = 0$	458,776	96.63%
$\Delta LCOM > 0$	9,874	2.07%
$\Delta LCOM < 0$	6,115	1.30%
Total	47,4765	100%

process. The first author detected the restructurings, and the last author checked all the list of deltas and list of restructurings. Whenever a doubt was raised, it was discussed to reach a consensus.

To analyze the evolution of the LCOM metric of a given class, we calculate the difference between the values of the metric from one release to another, where C is a class and R a release of the system on which the class exist: $\Delta LCOM = LCOM(C_{R+1}) - LCOM(C_R)$.

We consider three possible different outcomes for $\Delta LCOM$:

1. $\Delta LCOM = 0$ represents a stability situation of the class in which there was no change in the relationship between methods and attributes, captured by LCOM.

2. $\Delta LCOM > 0$ represents the increase of LCOM in the newer release, indicating a situation of increase of methods that do not share attributes in the class.

3. $\Delta LCOM < 0$ represents the decrease of LCOM, indicating the decrease of methods that do not share attributes.

Table 2 summarizes the result of applying the calculation of all $\Delta LCOM$s for all pairs of classes in adjacent releases. We can observe a stability in LCOM in most classes during the system evolution.

The situation in which ΔLCOM<0 is the one we are interested in, because it may indicate structural improvement in a class since there is fewer methods that do not share attributes. However, a small ΔLCOM may not be interesting from the point of view of an actual structural improvement. To determine if the ΔLCOM is significantly sharp, a threshold for a sharp ΔLCOM must be defined. To define that *sharpness* threshold, we use the Median Absolute Deviation (MAD) for selecting *outlier* ΔLCOM values, which is considered more robust than the absolute deviation of the mean or standard deviation to determine the presence of outliers[13]. MAD was calculated for the set D of values of $\Delta LCOM < 0 = \{d_1, \ldots, d_n\}$. Equation 1 shows the calculation of MAD. These outliers may indicate points in which classes had significantly sharp changes in their LCOM metric, and so more likely to be result of a anti-bloater class refactoring.

$$MAD(D) = median(\{|d_i - median(D)|\}) \tag{1}$$

On the 6,115 values of $\Delta LCOM < 0$, shown in Table 2, the median absolute value was 10, and the MAD value was 8. So, we consider the threshold equals the respective sum (18) to identify decreases in $\Delta LCOM$ that would be likely to be caused by a substantial class restructuring process. We found that the absolute $\Delta LCOM > 18$ occurred in 1,048 distinct classes. For those classes, where it occurred more than once, i.e., in more than one pair of releases, we have considered the greater absolute value.

The next step is to verify if actual restructuring has undergone in those classes where high variations of LCOM have occurred. We performed a qualitative manual inspection comparing the source code of the class in the two adjacent releases to find out if a restructuring has been carried out, similarly to the one performed for the *IndexWriter* class in Section 2. The verification if two versions of a same class in two consecutive releases has undergone an anti-bloater restructuring (Extract (Super-)class or Move method) is essentially based on the inspection if there is 1) a new class that has been created or a target class for moving methods and 2) either a instance variable in the new version of the original class referencing the respective class (Extract class/Move method), or original methods that were "*removed*" and may have been moved to other classes, or an `extends` clause in the created class extending the original class (Extract Superclass). Moreover, methods and variables should have been moved from the original class to the target class.

The manually inspected and evaluated points during the analysis were: 1) the nature of the moved methods: the percentage of simple get/set/add/update/remove, compared to more elaborated algorithms; 2) the target class where the methods were moved to; 3) the kind of coupling was defined from the source class to the target class: the percentage of association with instance variables, compared to inheritance; 4) the time of creation of the target class: the pre-existence or not of the target class and the percentage of gradual moving of methods; 5) what was the impact in the LCOM metric of the target class.

Due to the large number of classes (1,048), we extracted a random sample to conduct the manual inspection. The sample size was defined based on the following criteria:

1. We target a sample of 10% of the classes to make feasible the manual inspection.
2. At least 10% of the classes of each system, in which ΔLCOM occurred above the threshold, were considered. This criterion aims at guaranteeing that all systems were represented in the sampling.
3. In systems with only few classes, we opt to select more than 10% of them, so we select at least an arbitrary number of classes per system. This criterion aims at guaranteeing that systems with a few number of classes be represented with at least an arbitrary number of classes. We choose at least five classes for the sake of maintaining the sample in a manageable size.

These criteria favored a stratified sampling choosing classes from all eight analyzed systems. In the end, we sampled 107 ($\approx$ 10%) classes for manual evaluation, out of the 1,048 detected with highly negative $\Delta LCOM$. Tomcat had the highest number of classes (38), and JMeter, Log4J and JavaCC, the lowest number of classes (5).

3.3 | Manual analysis findings

First of all, we could confirm that the 107 classes had undergone actual restructuring. A qualitative analysis was performed comparing the class source code between the releases, in which the LCOM had decreased, to determine what types of restructuring strategies were adopted to restructure the respective classes. In two classes, the authors discussed the classification. The class *NioEndPoint* from Tomcat was actually confirmed to be extracted into *AbstractEndPoint* as an *Extract Class*. The only correction on the original list was that the class *Http11Protocol* was extracted into *HttpBaseProtocol* with an *Extract SuperClass* and not with an *Extract Class*.

We classified the found restructurings into three dimensions:

1. Type of dependency between classes: *Instance variable* or *Inheritance*. When an anti-bloater class restructuring is applied, the original class is split and a dependency relationship is generated between the new classes (source and target classes). When the anti-bloater class restructuring is similar to an Extract Class or Move Method, an instance variable with the same type of the target class is created in the source class. When the restructuring is an Extract Super-class, the dependency is made by the inheritance relationship.
2. The time at which the target class was created and the methods moved: *Immediate* or *Gradual*. This is a very distinguishing analysis of this work, because the vast majority of studies around refactoring detection consider those refactoring operations atomic in time, i.e., an immediate refactoring occurring in a single commit. However, we could observe that many refactoring operations are gradual in time, i.e., they are conducted partially, duplicating and deprecating code, to finish the refactoring operation in the future, spanning different releases. To determine if the strategy of moving the methods to the target class occurs immediately or gradually, we analyzed whether the target class already existed prior to the removal of the methods from the original class. If the class already existed it may be a Move Method, but can also be an Extract Class that had taken more than one release to get completed.
3. Percentage of get()/set() methods moved: To determine if there is likelihood for the choice of this kind of method, the percentages of get()/set() methods moved from the source class to the target class were analyzed.

Table 3 shows the result of the analysis conducted on the sample of 107 classes of eight systems.

With respect to the type of dependency between the classes, a predominance of instance variable to access the extracted class (and respective methods) was identified: 76.93%, against 23.16% of inheritance .

With respect to the time required to perform the restructuring operation, although restructuring is more likely to be concluded in the same release (61.53%), interestingly, still 38.47% of restructuring operations spanned more than one release. With respect to the nature of moved methods, we observe a tendency to choose public getter/setter methods. Regarding to other moved methods, we observed that a large part of

Dependency	Time	Prevalence	Ratio - getter/setter
Instance variable	Gradual	30.78%	58%
Instance variable	Immediate	46.15%	61%
Inheritance	Gradual	7.69%	42%
Inheritance	Immediate	15.38%	50%

TABLE 3 Qualitative Analysis Results

them, despite not having names get () and set (), they assume similar roles in modifying the attribute values, such as add(), read(), update(), so these kinds of simple methods are more likely to be moved, possibly because of their simplicity, and thus, possibly less impact in disrupting the code.

3.4 | Discussion on the time frame to refactor

The advantages and disadvantages of each type of restructuring strategy regarding to the time to proceed with it, and the respective impact on quality metrics are discussed below.

3.4.1 | Immediate restructuring

We observed that the immediate restructuring strategy identified in the qualitative analysis seems to have the following advantages:

- Immediate impact on structural and size metrics of the source class: the methods are removed from the source class at the same time that the target class is created, reducing the number of methods and increasing cohesion of the source class.
- It does not generate code duplication because the moved methods are not duplicated.

In contrast, the immediate restructuring strategy poses the following disadvantage:

- It causes direct and indirect impact on other classes of the system. The impact can be generated on classes that somehow depend on resources of the original class. For example, if a superclass method that is used by the subclass is moved to a new class during restructuring, a link must be created between the subclass and the new class, so that method can be accessed. This dependency relationship can be either in terms of instance variable binding, inheritance, or even code duplication.

Figure 3 shows an example of immediate restructuring performed on the **BaseResourceCollectionWrapper** class of the Ant system.

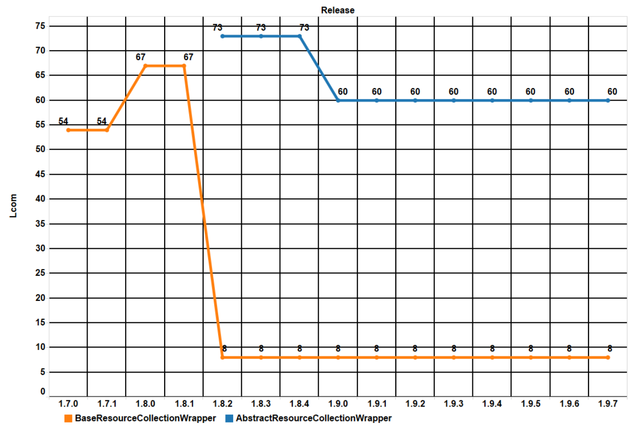

FIGURE 3 An example of immediate restructuring of BaseResourceCollectionWrapper

We observe that the impact generated on the source class is immediate, and occurs in the same release in which the class **AbstractResourceCollectionWrapper** is created. However, the target class was created with worse LCOM compared to the source class. This is partly due to the fact that the new class is also receiving brand new methods that would have been inserted in the source class if restructuring was not carried out.

3.4.2 | Gradual restructuring

The gradual restructuring strategy identified in the qualitative analysis has the following advantage:

- Time to adapt: By gradually removing methods from the source class, developers who consume resources of this class can also adapt gradually the code to use the methods of the target class and to stop using the methods marked as deprecated from the source class. This strategy minimizes the impact on classes that depend directly or indirectly on the original class.

In contrast, the strategy of gradual restructuring has the following disadvantage:

- Temporary duplicate code: the deprecated methods in the original class will be temporarily duplicated in both classes. During this time, any maintenance activity involving the source class can become more complex.

Figure 4 shows gradual restructuring of the **Redirector** class of the **Ant** system.

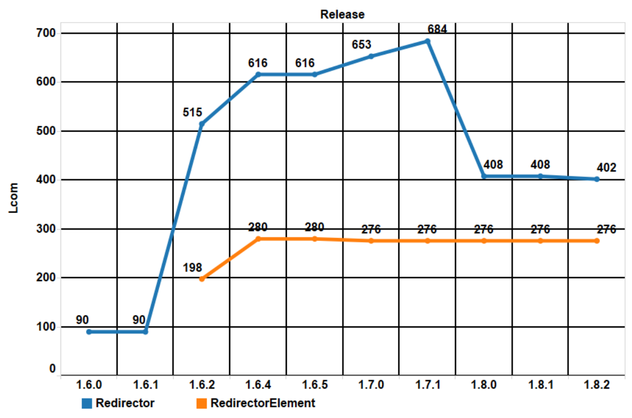

FIGURE 4 Gradual restructuring of Redirector

We observe that one release after its creation, the LCOM value of class *RedirectorElement* increases because of insertion of methods copied from *Redirector*. With respect to the size of the time frame that the methods remain duplicated, we observe that it might depend on how much time developers need to assure that the methods can be removed safely without affecting clients.

4 | PREDICTIVE MODEL FOR RESTRUCTURING

In the previous section, we could detect classes that had undergone important anti-bloater class restructuring. The identification of those classes was possible because we could compare the LCOM metrics of respective classes between two consecutive releases. However, considering that we aim at identifying classes that are prone to restructuring without observing its past history, but just its current snapshot, we may need an alternative that can predict the proneness to restructuring observing only its current metrics. So, we propose to predict when a class is likely to be restructured training a model with the selected 1,048 classes of the previous section that had undergone such restructuring. Our hypothesis is that structural and size metrics can produce an adequate predictive model for classes with restructuring opportunities.

4.1 | Method

To determine if there is any pattern in other structural and size metrics to predict the situations identified in the previous analyzes, a predictive decision tree model was constructed using Scikit-learn[14], which uses an optimised version of the CART (Classification and Regression Trees) algorithm. Decision Trees are a non-parametric supervised learning method used for classification and regression and CART is very similar to C4.5[15], but it differs in that it supports numerical target variables and does not compute rule sets. CART constructs binary trees using the feature and threshold that yield the largest information gain at each node. Moreover, we choose the decision tree model because we can learn and assess the conditions for such decision more intuitively.

The following steps are performed to construct the model.

	Fold 0	*Fold 1*	*Fold 2*	*Fold 3*	*Fold 4*	*Fold 5*	*Fold 6*	*Fold 7*	*Fold 8*	*Fold 9*	*Mean*	*Std*
F1[1]	0.94118	0.95364	0.94667	0.92715	0.96689	0.94194	0.95425	0.96000	0.94118	0.97333	0.95062	0.01308
Precision	0.91139	0.93507	0.93421	0.89744	0.93590	0.89024	0.91250	0.93507	0.90000	0.94805	0.91999	0.01903
Recall	0.97297	0.97297	0.95946	0.95890	1.00000	1.00000	1.00000	0.98630	0.98630	1.00000	0.98369	0.01584

[1]F1-score of validation dataset is used to choose the best decision tree model.

TABLE 4 Metrics for validation phase of the decision tree model.

1. We randomly sample 1,048 classes that were not restructured, i.e., classes with $\Delta LCOM = 0$ in all pairs of releases to serve as negative cases. So, the dataset is comprised of 2,096 classes: 1,048 that has undergone restructuring (TRUE cases) and 1,048 that were not restructured (FALSE cases).

2. We collect structural and size metrics for the selected classes. The metrics were collected with CKJM extended [16], which collects 18 different metrics (WMC - *Weighted Methods per Class*, DIT - *Depth of Inheritance Tree*, NOC - *Number of Children*, CBO - *Coupling Between Classes*, RFC - *Response For a Class*, LCOM - *Lack of Cohesion in Methods*, Ca - *Afferent couplings*, Ce - *Efferent couplings*, NPM - *Number of Public Methods*, LCOM3 - *Lack of Cohesion in Methods 3*, LOC - *Lines Of Code*, DAM - *Data Access Metric*, MOA - *Measure Of Aggregation*, MFA - *Measure of Functional Abstraction*, CAM - *Cohesion Among Methods of Class*, IC - *Inheritance Coupling*, CBM - *Coupling Between Methods*, AMC - *Average Method Complexity*). Out of these 18 metrics, we trained a decision tree with 17 metrics. In particular, we do not use the metric LCOM because it had been used to define the target (non-restructuring or restructuring) as described in Section 3. A special characteristic of the tree algorithm is its ability to choose the relevant features and automatically discard the other. After training, the following metrics were considered relevant by the algorithm: WMC, AMC, CBM, and IC.

3. We split randomly the dataset in training, validation and test sets [17]. The training data is the subset used to train the decision tree and validation subset used to evaluate how well the trained model is doing. These two subsets representing 70% of all dataset and they are obtained by k-fold cross-validation ($k = 10, FoldSize \approx 146$). K-fold split dataset into k consecutive folds (shuffling by default) and each fold is then used once as the validation data while the $k - 1$ remaining folds form the training set [14]. The folds are made by preserving the percentage of samples for each class. At the end, the test dataset (30% of all dataset) is used to provide an unbiased evaluation of final model fit and it is only used once a model is completely trained. We also ensured the same proportion of restructured (TRUE) and not restructured (FALSE) classes in both training and test sets.

4. During the train and validation, we tuned the model by using the hyperparameter optimization [18]. The traditional way of performing hyperparameter optimization has been grid search (*GridSearchCV* [14]) which is simply an exhaustive searching through a manually specified subset of the hyperparameter space of a learning algorithm. In other words, many models are created/trained and the best one is selected. In this paper, we use F1-score to get the best one. In this step, the following parameters of the tree are used to create the models: 1) *criterion*: the function to measure the quality of a split; 2) *min_samples_split*: the minimum number of samples required to split an internal node; 3) *min_samples_leaf*: the minimum number of samples required to be at a leaf node; 4) *max_depth*: the maximum depth of the tree; 5) *max_features*: the number of features to consider when looking for the best split; 6) *random_state*: always defined as zero to obtain a deterministic behavior during fitting. Thus, these parameters were adjusted iteratively to find a optimal configuration for decision tree[2]. In the end, the following configuration was the best: $criterion = "entropy"$; $min\_samples\_split = 16$; $min\_samples\_leaf = 16$; $max\_depth = 4$; $max\_features = 13$; $random\_state = 0$. The metrics of each fold of the best model are shown in Table 4.

5. After getting the best model, we evaluate the model with test dataset. The Table 5, 6 and 7 reporting the performance in terms of *confusion matrix*, *F1-score*, *precision* and *recall*.

4.2 | Results

Figure 5 presents the decision tree generated after training the predictive model. Given the metrics of a class, the model returns if the class should be restructured (TRUE) or not (FALSE). In the decision tree, the nodes located at higher levels are the variables/metrics that have higher weight for the classification. The nodes (boxes) in decision tree have information of relevant variables of the model, for example: the first node on the top of Figure 5 indicates that WMC is the most relevant metric for the classification. At this point on the tree, we have 1466 samples and they are

[2] Sklearn Decision Tree -https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

equally distributed in each class (733 for each target). These samples are used to create other two subnodes, in this case the rule WMC $\leq$ 8.5 is defined by the CART algorithm as a threshold. Thus, in the left, we have a new subnode with 508 samples which WMC $\leq$ 8.5 (501 and 7 for non-restructured and restructured, respectively) and WMC still used as a metric/threshold to split the next subnodes. On the other hand, in the right, we have a new subnode with 958 samples which WMC > 8.5 (232 and 726 to target non-restructured and restructured). At this point, the metric AMC is used to split these 958 samples into the next sub-nodes. In other words, at this point in the tree, the AMC is more relevant than WMC.

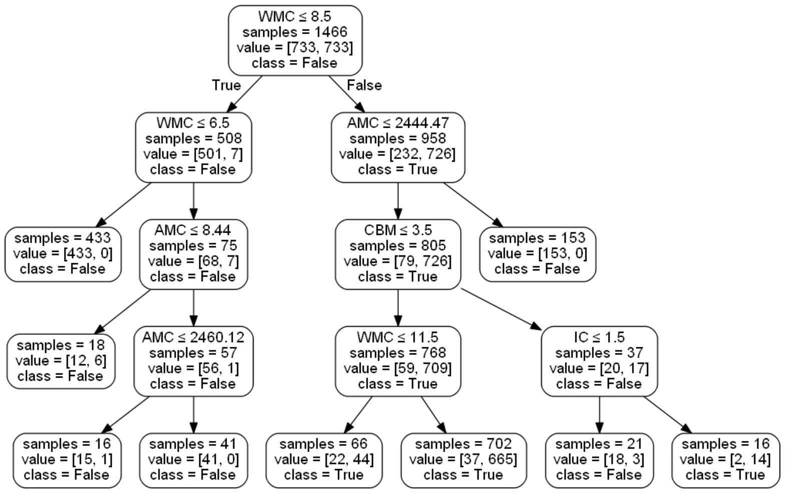

FIGURE 5 Restructuring decision tree

We present the patterns for recommending restructuring, i.e., leaf nodes labeled with TRUE.

The first pattern is related to the WMC (Weighted Methods per Class), which in CKJM extended is calculated simply as the number of methods of the class. Interestingly, empirical data obtained by Chidamber and Kemerer[7], more than 25 years ago, indicate that most of the classes have $0 \leq WMC \leq 10$. We can observe in the tree that classes with $WMC \leq 8.5$ tend to be predicted as *non-restructured*. By the root node, this single rule classify 508 samples ($\approx$ 34.6% out of 1,466). In the subnode, by analyzing the left nodes of the tree, 433 classes predicted as *non-restructured* are related to the low value of WMC (≤ 6.5). We also observe another metric used to predict classes as *non-restructured*: the metric AMC (Average Method Complexity) which measures the average method size for each class. The tree is able to predict 68 samples of non-restructured classes, with the inclusion of AMC and its combination with WMC. Thus, 501 (98.6% out of 508) samples of non-restructured classes are classified by the following rules: $(WMC \leq 6.5) \vee (6.5 < WMC \leq 8.5 \wedge (AMC \leq 8.44 \vee AMC > 2,460.12))$.

On the right side of the root node ($WMC > 8.5$), we have 65.3% (958 out of 1,466) of the training/validation dataset. At this node, the previous rule is complemented by the metric AMC ($\leq 2,444.47$). As the result of this combination, we have 232 classes predicted as *non-restructured* and 726 classes as *restructured*. But this single combination is not enough to classify the samples. Thus, the metrics CBM, WMC, and IC are included to the model by the algorithm. We observe that CBM is more relevant than WMC and IC. Note that $CBM \leq 3.5$ is used to guide which metric will be used on the path (WMC or IC). When IC is used and the threshold is less equal to 1.5, the samples are classified as *non-restructured*. On the other side ($IC > 1.5$), the samples are classified as *restructured*. Additionally, when WMC is used on the path, the samples are classified as *restructured*. Thus, by combining the metrics WMC, AMC, CBM, WMC and IC in specific thresholds, the tree classify 894 (93.3% out of 958) samples — 171 *non-restructured* and 723 *restructured*.

In general, on the training and validation step, our tree model reached a high ratio on the following metrics: F1-score ($\approx$ 95%), Precision ($\approx$ 91%), and Recall($\approx$ 98%), as seen the Table 4.

Additionally, Table 5, 6 and 7 presents the results of applying the decision tree model on the test dataset. The set with of 630 samples was used to evaluate the F1-socre, Precision, Recall and Accuracy, and we also observe similar values ($\approx$ 94%). We also evaluate the individual influence of each systems (see Table 7), that may exert an impact on the tree model. We observe that although there is some variation, that variation may not impair the accuracy of the worst result, which is still high. So, the produced model is sufficiently accurate to recommend classes prone to restructuring.

		Predicted	
		FALSE	TRUE
Real	**FALSE**	287	28
	TRUE	7	308

TABLE 5 Confusion Matrix — Test dataset

	F1-score	*Precision*	*Recall*	*Support*
FALSE	0.942529	0.976190	0.911111	315
TRUE	0.946237	0.916667	0.977778	315
accuracy	0.944444	0.944444	0.944444	—-
macro avg	0.944383	0.946429	0.944444	630
weighted avg	0.944383	0.946429	0.944444	630

Precision can be seen as a measure of a classifier's exactness. Recall is a measure of the classifier's completeness. F1-score is a weighted harmonic mean of precision and recall. Support is the number of occurrences of the class. Weighted avg, used on an unbalanced dataset, is calculated accordingly to the number of true instances (i.e., $X = \#True \div \#All, weighted_{avg} = X \cdot Class_{true} + (1 - X) \cdot Class_{false}$). Macro avg is similar to Weighted avg, but the weighted equal for all classes (i.e., 0.5). For more information, see Scikit-learn documentation.

TABLE 6 Metrics of Confusion Matrix — Test dataset

		F1-Score	*Precision*	*Recall*	*Accuracy*	*Support*
Log4j	**macro avg**	0.938889	0.947368	0.937500	0.939394	33
	weighted avg	0.939057	0.945774	0.939394		
Xerces	**macro avg**	0.912562	0.924641	0.905714	0.916667	60
	weighted avg	0.915719	0.919657	0.916667		
Lucene	**macro avg**	0.975090	0.980000	0.971429	0.975904	83
	weighted avg	0.975795	0.976867	0.975904		
JMeter	**macro avg**	0.866071	0.866071	0.866071	0.866667	30
	weighted avg	0.866667	0.866667	0.866667		
Ant	**macro avg**	0.907126	0.905242	0.909647	0.909091	55
	weighted avg	0.909337	0.910191	0.909091		
Javacc	**macro avg**	1.000000	1.000000	1.000000	1.000000	6
	weighted avg	1.000000	1.000000	1.000000		
Jetty	**macro avg**	0.956739	0.955952	0.958820	0.956897	116
	weighted avg	0.956964	0.958313	0.956897		
Tomcat	**macro avg**	0.951352	0.954261	0.951580	0.951417	247
	weighted avg	0.951345	0.954410	0.951417		

TABLE 7 Metrics of test dataset by system

5 | SURVEY WITH DEVELOPERS

After developing a model that allows to identify classes with restructuring opportunities, a survey with the developers of the studied systems was conducted to evaluate the proposed recommendations. In this context, the survey will provide the perception of developers on the topic, measuring their level of interest and engagement in restructure a class to potentially improve system modularity.

The main objectives of the survey are to:

- Complement the exploratory study of Section 3, analyzing the strengths and drawbacks of the recommendations to come up with possible improvements for the restructuring strategies.

- Perform another kind of assessment of the model concerning the identification of classes with opportunity for restructuring. Based on the developers' responses, we aim at evaluating whether the recommended classes by the predictive model are appropriate sites for restructuring.

- Evaluate the format of the recommendation guides in terms of difficulty level for understanding the recommendations.

- Identify other characteristics in the classes and in the system design that can hinder the restructuring process.

5.1 | Survey design

A survey was proposed to evaluate the recommendations of restructurings to the developers of the eight studied systems: Ant, Log4j, Jetty, Lucene, Xerces, JMeter, Tomcat and Javacc.

The restructuring recommendations were presented to the developers by creating issues on the bug trackers of the systems: Jira, Bugzilla, GitHub. We randomly selected seven classes that the predictive model from the previous section has indicated to be prone to restructuring. Then, for each class we opened a issue in the respective system suggesting the restructuring. Since we were interested in the exploratory results of the survey, we indeed continued the discussion of the issue with those developers that showed interest in the recommendation, either being positive or presenting any concern on the recommendation. Fifteen developers responded the issues and some of them continued the discussion as shown below. The respondents are made anonymous in this paper. Out of the fifteen developers, twelve are Project Management Committee members, and also Committers. About the other three respondents, *dev_ant_2* is a collaborator that opens pull requests, *dev_jetty_1* contributed 701 commits by February 2021, and *dev_jmeter_1* is a collaborator in JMeter, but has committer role in other Apache projects.

Following, we show the systematic process to generate a issue description (or mail message) that has also the role of a restructuring guide. The issue has three parts:

5.1.1 | Object class

Each issue is associated with one class, so we introduce the name of the class in which the restructuring opportunity was identified. In the following example, the selected class is the *SegmentInfos* of the Lucene system. The preamble of the respective issue is:

- *"Hello everyone. I was analyzing the modularization of some classes, and I identified that the class SegmentInfos has an opportunity for cohesion improvement."*

5.1.2 | Past similar situation

The second part of the issue is an example of another class in the system that had been restructured. We selected that similar class in the set of restructured classes presented in Section 4, and the choice was taken comparing the proposed suggestion of restructuring with the observed restructuring in the example class. We may use the same example class for several classes. In the current example, we show a reference to a similar change in the *IndexWriter* class that could motivate the restructuring in the *SegmentInfos*. In both classes, a configuration class was extracted from the original class.

- *"The class IndexWriter was in the same situation and solved as follows: The IndexWriterConfig class was created, and several get () and set () methods that were used only to set the class parameters were moved from IndexWriter to IndexWriterConfig. The new class was then accessed through an instance variable in IndexWriter. This strategy has cleaned and improved IndexWriter cohesion. "*

5.1.3 | Restructuring strategy

The last part of the issue presents the strategy suggested to perform the restructuring. In the example, the recommendation is the creation of a new *SegmentInfosConfig* class (similar to the *IndexWriterConfig*) to receive the configuration methods of *SegmentInfos*, and how the new class should be referenced from the original class. In this example, the dependency is made through an instance variable.

- *"With this in mind, I would recommend creating a new class: SegmentInfosConfig, and moving the following methods: getLastCommitGeneration, getLastCommitSegmentsFileName, getSegmentsFileName, getNextPendingGeneration, getId, getVersion, getGeneration, getLastGeneration, setInfoStream, getInfoStream, setNextWriteGeneration, setUserData, setVersion, getCommitLuceneVersion, getMinSegmentLuceneVersion from the SegmentInfos. Those parameters accessed by an instance variable in the SegmentInfos. Moreover, the orthogonality of the design would be enhanced. What do you think about that?"*

5.2 | Analysis of survey results

Table 8 presents the survey results for the recommendations of the selected classes. A total of 56 recommendations were performed (seven recommendations for each system), and 31 responses were obtained. Out of these 31 responses, 13 were classified as neutral. A neutral response means that the developer did not evaluate the recommendation, but instead raised a related discussion on the problem. The other 18 responses effectively evaluated the recommendations, whose results are in Table 8.

					Target Class		Strategy	
System	**Guide**	**Answers**	**Neutral**	**Non-neutral**	**Correct**	**Incorrect**	**Correct**	**Incorrect**
ant	7	7	0	7	6	1	6	1
log4j	7	1	0	1	1	0	1	0
Jetty	7	1	0	1	1	0	0	1
Lucene	7	7	5	2	1	1	0	2
Xerces	7	1	1	0	0	0	0	0
Jmeter	7	7	7	0	0	0	0	0
Tomcat	7	7	0	7	7	0	0	7
Javacc	7	0	0	0	0	0	0	0
Total:	56	31	13	18	16	2	7	11
%Total:	100%	55,36%	41,94%	58,06%	88,89%	11,11%	38,89%	61,11%

TABLE 8 Survey results.

Moreover, we conducted an open coding analysis on the response of the authors to extract the recurrent themes in the obtained answers[19]. Table 9 shows the result of the open coding analysis of the survey answers.

Evaluation	Theme	Occur.
positive	preventive restructuring strategy	2
positive	refactory opportunity localization and development of strategies	5
neutral	reuse and duplication of code	2
neutral	lack of knowledge about the architecture	4
neutral	backwards compatibility	2
neutral	build break	1
neutral	bugs with higher priority	2
neutral	guide format and additional information	2
negative	inadequate list of selected methods	2
negative	lack of analysis of internal classes	2
negative	insufficient number of methods	2
negative	age of the class and constant changes	2
negative	difficulty to use and loss of performance	3

TABLE 9 Thematic analysis of the answers.

In order to unveil the discussion on the themes, we will present some excerpts of the answers. To ensure developer anonymity, comments are anonymized. We follow the order of Table 9, showing the positive, neutral and negative responses, which include alternatives adopted to overcome the difficulties in the process of restructuring, as well as new possible restructuring strategies.

Preventive restructuring strategy. When performing the restructuring recommendation for the Project class of the Ant system the developer dev_ant_1 replied: *"I'm sure your changes would improve the quality of the *X* class but the same time it scares me".*

The main concern is related to the developers who use resources of this class and who would need to be informed about the change. Considering that concern, a gradual restructuring strategy was proposed instead of an immediate restructuring. That way the methods would be marked as *deprecated* as a warning to developers that they will be removed and that there are others to be used in place. On the proposal of this new strategy, the answers is: *"This strategy is a sound strategy if you expect users of your API to follow releases closely. Unfortunately this is not what we see with SystemX. Right now we are fielding a bug reported by somebody who is migrating from n to n+y - Version n has been superseeded by n+1 in 2005. This is not uncommon.This has led to us never removing any methods at all, no matter how long it has been deprecated".*

Given the difficulty in updating the developers about the changes made, and considering that removing the methods from the class is not a viable option, a **third modularization/restructuring** strategy is proposed. The strategy is the creation of a new class to receive the getter and setter methods that would be inserted into the *original* class in the future. In this way, although it is not possible to solve the current problem of the class, at least a greater lack of cohesion can be avoided in the future. This strategy was presented to the developer with the name of **preventive restructuring**, although it is not actually a restructuring but a modularization strategy to simulate a possible necessary future restructuring that would not be done because of the concerns on impact of user of the *original* class. When asking the developer if he would adopt this strategy even as a standard for creating new classes, the answer was: *"Absolutely"*.

Restructuring opportunity localization and development of strategies. The approach has shown to be quite promising with regard to the location of opportunities for restructuring. However, because not all suggested restructuring strategies have been evaluated as positive, some developers were willing to help formulate new strategies more appropriate to the needs and constraints of the system, recognizing the need for improvement in the classes pointed out by the approach. When evaluating a recommendation and having rejected the restructuring strategy the developer states: *"So yes I'd like to see a cleaner class, but it is a much more difficult task than it looks..."* This kind of comment show an interest on the part of the developers in strategies that promote modularization improvements, but sometimes a more complex solution that class splitting is required.

Reuse and code duplication. Reusing code through duplication is not uncommon in software systems. While copying the code can save time in the short term, this practice makes the maintenance process much more complex. When recommending the restructuring for a class, we had the following comment: *"this class derives from ***other class***. Every now and then we look at ***that*** class code to see what patches have been applied and update this class with them"*.

Developers somehow try to avoid duplication, but at the same time recognize the difficulty in avoiding this practice. The above example represents the classical problem of fixing duplicate code, i.e., the patch has to be applied in all duplicated parts.

Lack of knowledge about the architecture. Another recurring theme was about the role of classes in the system and the level of influence they exert on other classes. Upon receiving the restructuring recommendation for a class the developer replied as follows: *"I don't know this code, so every change is a major risk"*.

Analyzing the above feedback, the lack of knowledge about the system architecture significantly increases the difficulty and risk of restructuring a class. An interesting fact is that none of them cited the use of system documentation, which may indicate that current documentation does not meet the needs of developers. For instance, this kind of practical knowledge of the developers has proved of extreme value to improve recommendations in the sense that a recommendation should inform possible impacts on other classes, and preferably inform the risk level of restructuring.

Backward Compatibility. Because the subject systems have several available releases, one point that was evident in the discussions was the need for backward compatibility. During the presentation of the system's characteristics and discussion of the recommendation risks, the developer makes the following comment: *"having in mind that there are a lot of existing plugins and that backward compatibility is important within the project"*.

This comment demonstrates that backward compatibility imposes a major challenge for restructuring, and makes it difficult to evolve the system, especially with those changes that impact architecture. We could observe a lot of effort towards solving the problem of backward compatibility, evidencing opportunities for the implementation of automated solutions.

Breaking builds. On systems that have a very large dependency relationship with packages, APIs, and external plugins, the difficulty in avoiding build failures is evident. A simple removal of methods or a change in attribute type can generate build-breaking inconsistencies. When explaining about system integration problems, a developer states the following: *"We cannot remove any method used by it at all without breaking build files of people who rely on module X"*. The complexity of the environment upon which the system is built becomes evident. Techniques that aim for minimizing continuous integration breakages could be incorporated into restructuring strategies.

Bugs with higher priority. One point in the discussions that became clear was what changes should be prioritized for better use of available human resources. If the system features have errors, developers are expected to prioritize these fixes instead of modular improvements, e.g.: *"There are dozens of serious complex bugs open and that's where time should be spent"*.

In other words, the proposed restructuring are relevant for most developers, however they have lower priority compared to bugs and important new features, specially for systems with few developers.

Guide format and additional information. About the restructuring guide format, some developers have made suggestions. *"It would be best for you to make concrete proposals, basically an outline (I'd recommend relatively small chunks) of what kinds of changes you are thinking about. It doesn't even have to be a functioning patch for a first cut, just something concrete that people can comment on"*.

The way the recommendation is proposed impacts differently on developers. For some, the description of the strategy was sufficient, for others the code resulting from the restructuring should be presented.

Inadequate list of methods. On the restructuring recommendation for Lucene's *SegmentInfos* class the developer replied: *"It feels a bit wrong to me since commit gen, generation, version, etc. are not configuration parameters. I think SegmentInfos is fine as-is"*.

When analyzing the list of methods suggested to be moved to the new class, it was observed that although they have getter and setter names, and perform assignment and read operations, these operations were not being done for class attributes. Some of these methods were also overloaded, so more detailed semantic analysis would be required in this case.

Lack of internal class analysis. On the restructuring recommendation for the *Request* class of the Jetty system, the developer replied: *"Note also that Request already does have an inner instance that is kind of a RequestConfig. It is already passed a MetaData. Request instance which holds method, URI, version, headers and trailers of the request. Many of the methods you list are already handled by referencing this instance"*.

Number of insufficient methods. On the restructuring recommendation for Lucene's *IndexSearcher* class, the developer commented: *"In the case of IndexWriter there were so many getters and setters that we made this change due to the sheer number"*.

In fact, the proportion of getter and setter methods in the *IndexWriter* class prior to restructuring was higher than the *IndexSearcher* class. This observation indicates that it is necessary to define a threshold for the number of required methods to justify restructuring. Although, we tried to capture this feature using only classes that have high LCOM improvement, that option was not sufficient to capture this particular feature.

Class age and constant changes. About the age of the class and the modifications made over time the developers comment: *"well it is true that *X* is a very large somewhat ugly class that is carrying lots of history with it. It is the mutability aspect of the class that makes it rather difficult to simplify"*.

Based on this kind of comment, it is evident that the age of the class in the system presents difficulties for the implementation of restructuring strategies. By being constantly modified the complexity of the class increases significantly. One point to be improved in the approach is to identify the number of changes applied in the class over time, in order to help assessing the complexity and effort required to restructure.

Difficulty and loss of performance when creating a new class. On the fact that it is necessary to create a new class, a developer made the following comment: *"Its an additional class the user must worry about and more complicated than a POJO: I think it makes something hard to use"*. In fact, when creating a new class, more elaborated semantics should be introduced in the class, just than being a POJO, so this requires more complicated strategies.

6 | LESSONS LEARNED

The answers for our motivating questions can be discussed after these three studies.

Motivating question 1 (which): Is there a typical situation for a class, in terms of its quality structural metrics, that developers can use to decide restructuring that class?

Based on the analysis of evolution of the LCOM metric, restructurings can be identified in the points of significant variation in the metric value. Based on the result of the predictive model that identified patterns involving other size and structural metrics, we observed a strong association between the values assumed by some metrics and the class restructuring application. According to the identified patterns, the typical situation chosen for restructuring is mainly related to the complexity of the methods (i.e., WMC, AMC).

Given the accuracy of the model when locating these classes, and the perception of developers, we found that the analysis of the metrics is a reasonable proxy to determine if a class meets the requirements to be restructured. Nonetheless, we also observe that some more qualitative constraints should also be considered, because they can hinder the ability of developers performing changes in more intricate classes.

Motivating question 2 (how): Is it possible to detect a more appropriate strategy for restructuring a class?

Based on the result of the exploratory study of the identified restructurings that has shown that a large part of moved attributes/methods corresponds to simple getters/setters, and also together with the survey results, three restructuring strategies were detected. The first two: *gradual restructuring* and *immediate restructuring* were identified by comparing the source code of the classes that underwent significant changes in the LCOM metric. These strategies are related to the movement of getter and setter methods, for classes that perform a configuration role. Such strategies help to improve modularity by moving the configuration role to a separate class, leaving only the methods related to the core functionality of the source class. The way that methods are removed from the source classes has advantages and disadvantages, showing that the application of these strategies depends on the situation of the system. If several classes depends on the interface to be changed, then a gradual strategy seems to make sense, whereas if the interface is used more locally, an immediate restructuring can take place and complete the task more rapidly. The third strategy was identified through the survey results carried out with the developers, and it is a form of preventive restructuring that aims to avoid further degradation of the class. Therefore, we found that it is possible to identify restructuring strategies, but in order to determine if those strategies are suitable for a class, a deeper knowledge about the system architecture is typically necessary.

7 | THREATS TO VALIDITY

In this section, we report some identified threats to validity[20].

Conclusion validity. This kind of threat is about how sure we can be that the treatment is really (statistically) related to the actual outcome. Since some of the analyzes were done by sampling, there is a risk of some sample defect in which the sample does not represent the population. To mitigate this risk, reasonably large and balanced samples were selected. So, the representativeness of the systems was addressed in all stages of the analysis. In our case, we mitigate this threat with a reasonably large sample of systems and classes that were analyzed in Section 4. Moreover, in Section 5 there was also a significant ground-truth with 2096 instances.

Also, to identify whether there was a restructuring in a particular class, it was necessary to establish a threshold from which the variation of LCOM could be considered significant. In the end, that threshold is arbitrary. To mitigate this threat, the calculation of the DAM (Absolute Diversion of the Median)[21] was performed. This calculation presents a more reliable result than the traditional standard deviation being less sensitive to the presence of outliers[13].

Internal validity This kind of threat is about how sure we can be that the treatment actually caused the outcome. There can be other factors that have caused the outcome, factors that we do not have control over or have not measured. In our case, we need to reflect on if a sharp variation on LCOM actually is induced in classes that had undergone relevant restructuring. Although, this variation in LCOM is actually a proxy, we mitigated this threat manually inspecting those 107 classes and actually verifying that non-negligible restructuring has taken place. Moreover, we need to reflect on how that list of restructured classes can also be used to train a prediction model that can actually identify other classes that are also prone to restructuring. In order to mitigate this threat, we decided to conduct the survey with actual developer to better understand how effective those predictions would be in practice. Internal validity is related to other factors that may cause the outcome. In fact, our survey results helped to show that despite the very accurate prediction model for identifying classes for restructuring, the model is not as accurate in real world as shown by precision/recall tables, exactly because there are more factors related to software design that impact the restructuring decision. We recognized that threat in our conclusions, showing that although useful the model predictions require a further analysis step by the expert.

Construct Validity. This kind of threat is about the relation between the theory behind the study and the observation. During the manual inspection of the classes in Section 3, we aimed at finding if an Extract (Super-)Class or Move Method had taken place. This manual inspection may be challenging and error-prone because the target class for methods may have been created in previous releases (for the only purpose of moving those methods, or not), and inducing a confusion on determining if we have an Extract (Super-)Class or Move Method. Fortunately, the impact of this type of misclassification is small, since the most important observation is the identification if it is an anti-bloater restructuring or not, and all these kinds of restructurings induces anti-bloating. Other errors may have occurred during the identification of restructurings due to possible human misdetection. We tried to mitigate this threat organizing the identification in two parts: the first author identified the restructurings and the last author reviewed them. In case of doubt, a discussion has taken place to reach a consensus.

External Validity. This kind of threat is related to whether we can generalize the results outside the scope of our study. Our data come from a subset of software systems of the Apache ecosystem. Although, we have some diversity on the selected systems, we agree that other sample of systems could have a more diverse set of characteristics, that would indicate other possible alternatives and results.

8 | RELATED WORK

8.1 | Refactoring prediction

RIPE (Refactoring Impact PrEdiction) is a technique that estimates the impact of refactoring operations on software quality metrics[22], aimed to help developers choosing the most appropriate refactoring strategy. The study evaluated the impact of 12 refactoring strategies on 11 software quality metrics. An accuracy of 38 % was obtained for estimating the impact of the refactoring strategies. In our approach, we not aim at estimating the impact of refactoring strategies, but at defining a more accurate approach to recommend anti-bloater class restructuring. The ground-truth of restructurings of our work is constructed from the variation in a structural metric (LCOM) generated by the restructurings/refactorings. So, RIPE aims at estimating the immediate impact of refactoring on the classes involved, and does not take into account the history of class changes to determine if this is an expected time to apply refactoring. Whereas, in our work, we conducted an evolution analysis of the classes in terms of the variation of a structural metric, so our ground-truth was designed with classes that reached such a condition that lead to their restructuring.

Software metrics have also been used to find refactoring recommendations aimed at helping developers to improve modularity[23,24,25,26,27]. Ratzinger et al. present an empirical study to determine what software evolution characteristics can be used to predict the need to apply refactoring in the next 2 months of development[28]. As a result, the study identified that software metrics such as LOC (Lines of Code) and metrics that indicate code modifications such as the Quantity of Added or Removed Lines can be used to develop models to predict the need for refactoring. These studies focus on quantitative metrics only, whereas in our work, besides the proposal of a predictor, a qualitative analysis of the changes was conducted to determine if a restructuring has occurred, which type of strategy was used, and the perception of developers on similar changes.

8.2 | Structural change evaluation

The problem of identifying modularity changes during the software evolution was also investigated by Antoniol et al.[29]. They propose an automatic approach to identify cases of possible refactorings based on vector space cosine similarity. ARCADE(Architecture, Change, And Decay Evaluator) was proposed[30] to evaluate changes in the software architecture. The architectural changes were evaluated using metrics, such as, a2a (architecture-to-architecture) that measures the distance between two architectures. As a result the study demonstrated that the version numbering system is not a good indicator of architectural changes. They also observed that architectural changes tend to occur in earlier versions of the systems. Differently from our study, these studies did not conduct a qualitative analysis on the refactorings to investigate how structural changes could indicate undergone refactoring operation.

The relation between the strategies of refactoring and the changes in software metrics has been widely discussed[31,32,33,34]. The comparison of the metric values before and after refactoring enables measuring the effects in the software quality[35]. Du Bois et al. present an empirical study aimed to identify under what circumstances refactoring brings improvements to cohesion[6]. Guides were proposed to apply different strategies of refactorings. The application of guides was evaluated with the Apache Tomcat. A total of 20 refactoring opportunities were identified in 12 classes. The identification of the opportunities was made choosing classes with cohesion problems, and among those classes, those that met the requirements for application of the guide were chosen. Differently, our approach proposed guides based on a qualitative analysis of when classes had a significant change in LCOM. Therefore, our proposed guides uses past refactorings as examples.

Bavota et al. propose a method to identify *Extract Class* opportunities based on graph theory and a combination of semantic and structural measures[36]. The graph edges represent the value of the metric SSM (Structural Similarity between Methods).Others studies were proposed to identify and apply *Extract Class* and *Move Methods* using clustering and game theory[37,38,39,40,41]. In contrast to our approach, they do not consider temporal analysis of the LCOM to determine when classes present the opportunities for refactoring.

8.3 | User perception of refactorings

One of the main challenges to recommend a refactoring strategy is to understand the perception of the developer and what leads to refactoring. Silva et al. studied the main motivations for developers to apply refactorings[42]. A survey conducted with developers found that in 75% of cases, the motivation to refactor a class is more related to insertion of new features and bug fixes than to structural problems in the code. Other study[43] investigated the relation between refactoring and changes, and observed that developers tend to refactor to improve cohesion when they are implementing new features, whereas when they are fixing bugs, refactorings are aimed at maintainability and comprehensibility.

Kim et al. presented a study on the challenges and benefits of refactoring[44]. The purpose of this study is to better understand the definition of refactoring, benefits and challenges from the perspective of developers. The study found that in practice refactoring does not always preserve the behavior of the system; that developers understand that refactoring involves costs and risks, and that it is not always possible to preserve system behavior by modifying the source code. The quantitative analysis showed that the refactoring process brings benefits, such as reduction of dependencies between classes, and reduction of defects. We could observe similar perceptions in our study.

Tempero et al. presented the barriers and reasons that lead developers to not refactor[45]. Their results present some similarities with our approach and the results of the survey. The lack of knowledge and the risk to introduce new faults in the system are highlighted by developers, i.e. refactoring recommendations must provide knowledge about the involved classes, in order to provide developers with a better understanding of the impacts of the refactoring. The results in these works are consistent with our survey, in the sense that recommendation on restructuring a class should address several contextual elements related to the overall design.

9 | CONCLUSIONS

In this work, an approach to identify class restructuring opportunities was presented. The strategies on how to perform those operations were defined, two of which were identified through an exploratory analysis of the evolution of size and structural metrics, and the third through a survey with the developers. Based on our results, we confirmed the adequacy of a predictive model for recommending anti-bloater class restructuring. Nonetheless, the task of class restructuring is complex and have several subtleties that our qualitative survey has raised and provided insights for improvement of the recommendations. Some lessons have been learned: 1) Better fine-tuned thresholds can avoid recommend classes whose impact is not evident, i.e., low number of extracted attributes and methods does not justify the risk of refactoring; 2) The history of the class can provide important data, such as, code churn, to avoid unnecessary recommendations, i.e., high complexity classes with a history of intensive changes are less likely to be restructured, and thus should not be recommended; 3) Recommendation systems should learn more about the design and context of the object systems in order that recommendations make more sense to expert architects.

During the discussion with the developers some proposals for future work were raised: necessity of an environment for simulation of restructuring to understand the impact; reinforced learning applied to restructuring actions; identification of patterns for system-specific refactoring activities. Moreover, we suggest the investigation on how commit-oriented refactoring detection tools[11,12] could be adapted to help the replication of our work with an automated detection approach.

Acknowledgments. We would like to thank the participants that responded the survey. The underlying data used in this study is open, anonymous and available[46]. We also would like to thank the Brazilian Funding Agencies CAPES, CNPq, and FAPEMIG for partially supporting this research.

References

1. Kannangara S, Wijayanayake W. An empirical evaluation of impact of refactoring on internal and external measures of code quality. *arXiv preprint arXiv:1502.03526* 2015.

2. Lehman MM, Ramil JF. Towards a theory of software evolution - and its practical impact. In: International Symposium on Principles of Software Evolution. ; 2000: 2–11

3. Fowler M, Beck K. *Refactoring: improving the design of existing code*. Addison-Wesley Professional . 1999.

4. Mäntylä MV, Lassenius C. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering* 2006; 11(3): 395–431. doi: 10.1007/s10664-006-9002-8

5. Sobrinho EVP, De Lucia A, Maia MA. A Systematic Literature Review on Bad Smells–5 W's: Which, When, What, Who, Where. *IEEE Transactions on Software Engineering* 2021; 47(1): 17-66. doi: 10.1109/TSE.2018.2880977

6. Du Bois B, Demeyer S, Verelst J. Refactoring-improving coupling and cohesion of existing code. In: IEEE. ; 2004: 144–151.

7. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 1994; 20(6): 476–493.

8. Gupta BS. A critique of cohesion measures in the object-oriented paradigm. Master's thesis. Michigan Technological University. Department of Computer Science, Michigan Technological university: 1997.

9. Srinivasan KP, Devi T. A complete and comprehensive metrics suite for object-oriented design quality assessment. *International Journal of Software Engineering and Its Applications* 2014; 8(2): 173–188.

10. Gélinas JF, Badri M, Badri L. A Cohesion Measure for Aspects.. *Journal of object technology* 2006; 5(7): 75–95.

11. Tsantalis N, Mansouri M, Eshkevari LM, Mazinanian D, Dig D. Accurate and efficient refactoring detection in commit history. In: International Conference on Software Engineering. ACM; 2018: 483–494.

12. Silva D, Silva dJP, Santos G, Terra R, Valente MT. RefDiff 2.0: A Multi-language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* 2020: 1–17.

13. Leys C, Ley C, Klein O, Bernard P, Licata L. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology* 2013; 49(4): 764–766.

14. Pedregosa F, Varoquaux G, Gramfort A, et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 2011; 12: 2825–2830.

15. Quinlan JR. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. . 1993.

16. Spinellis D. Tool writing: a forgotten art? (software tools). *IEEE Software* 2005; 22(4): 9-11. http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/.

17. Shalev-Shwartz S, Ben-David S. *Understanding machine learning: from theory to algorithms*. Cambridge University Press . 2014.

18. Bisong E. *Building Machine Learning and Deep Learning Models on Google Cloud Platform: A Comprehensive Guide for Beginners*. Apress . 2019.

19. Cruzes DS, Dyba T. Recommended Steps for Thematic Synthesis in Software Engineering. In: International Symposium on Empirical Software Engineering and Measurement. ; 2011: 275-284.

20. Feldt R, Magazinius A. Validity Threats in Empirical Software Engineering Research - An Initial Survey. In: SEKE. Knowledge Systems Institute Graduate School; 2010: 374–379.

21. Huber PJ. *Robust statistical procedures*. SIAM . 1996.

22. Chaparro O, Bavota G, Marcus A, Di Penta M. On the impact of refactoring operations on code quality metrics. In: IEEE. ; 2014: 456–460.

23. Demeyer S, Ducasse S, Nierstrasz O. Finding refactorings via change metrics. In: . 35. ACM. ; 2000: 166–177.

24. Ouni A, Kessentini M, Sahraoui H, Inoue K, Deb K. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2016; 25(3): 23.

25. Vidal SA, Marcos C, Díaz-Pace JA. An approach to prioritize code smells for refactoring. *Automated Software Engineering* 2016; 23(3): 501–532.

26. Bavota G, De Lucia A, Di Penta M, Oliveto R, Palomba F. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software* 2015; 107: 1–14.

27. Ounia A, Kessentinib M, Sahraouic H, Cinnéided MÓ, Debe K, Inouea K. A Multi-Objective Refactoring Approach to Introduce Design Patterns and Fix Anti-Patterns. In: North American Search Based Software Engineering Symposium. ; 2015.

28. Ratzinger J, Sigmund T, Vorburger P, Gall H. Mining software evolution to predict refactoring. In: IEEE. ; 2007: 354–363.

29. Antoniol G, Di Penta M, Merlo E. An automatic approach to identify class evolution discontinuities. In: IEEE. ; 2004: 31–40.

30. Le DM, Behnamghader P, Garcia J, Link D, Shahbazian A, Medvidovic N. An empirical study of architectural change in open-source software systems. In: IEEE Press. ; 2015: 235–245.

31. Simon F, Steinbruckner F, Lewerentz C. Metrics based refactoring. In: IEEE. ; 2001: 30–38.

32. Steidl D, Eder S. Prioritizing maintainability defects based on refactoring recommendations. In: ACM. ; 2014: 168–176.

33. Szőke G, Nagy C, Fülöp LJ, Ferenc R, Gyimóthy T. FaultBuster: An automatic code smell refactoring toolset. In: IEEE. ; 2015: 253–258.

34. Simons C, Singer J, White DR. Search-based refactoring: Metrics are not enough. In: Springer. ; 2015: 47–61.

35. Kataoka Y, Imai T, Andou H, Fukaya T. A quantitative evaluation of maintainability enhancement by refactoring. In: IEEE. ; 2002: 576–585.

36. Bavota G, De Lucia A, Oliveto R. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software* 2011; 84(3): 397–414.

37. Bavota G, Oliveto R, De Lucia A, Antoniol G, Gueheneuc YG. Playing with refactoring: Identifying extract class opportunities through game theory. In: IEEE. ; 2010: 1–5.

38. Fokaefs M, Tsantalis N, Stroulia E, Chatzigeorgiou A. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software* 2012; 85(10): 2241–2260.

39. Bavota G, Oliveto R, Gethers M, Poshyvanyk D, De Lucia A. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering* 2014; 40(7): 671–694.

40. Murphy-Hill E, Black AP. Breaking the barriers to successful refactoring: observations and tools for extract method. In: ACM. ; 2008: 421–430.

41. Silva D, Terra R, Valente MT. Recommending automated extract method refactorings. In: ACM. ; 2014: 146–156.

42. Silva D, Tsantalis N, Valente MT. Why we refactor? confessions of github contributors. In: ACM. ; 2016: 858–870.

43. Palomba F, Zaidman A, Oliveto R, De Lucia A. An Exploratory Study on the Relationship Between Changes and Refactoring. In: ICPC '17. International Conference on Program Comprehension. IEEE Press; 2017; Piscataway, NJ, USA: 176–185.

44. Kim M, Zimmermann T, Nagappan N. A field study of refactoring challenges and benefits. In: ACM. ; 2012: 50.

45. Tempero E, Gorschek T, Angelis L. Barriers to refactoring. *Communications of the ACM* 2017; 60(10): 54–61.

46. Machado JPL, Sobrinho EVP, Maia MA. A Replication Package For The Paper "Anti-bloater Class Restructuring: An Exploratory Study". https://doi.org/10.5281/zenodo.4670461; 2021

How to cite this article: J.P. Machado, E.V. Sobrinho, and M.A. Maia When and how to split classes: an exploratory study, *Journal of Software: Evolution and Process, 2020;00:xx–xx*, 2020.