Deep Learning Anti-patterns from Code Metrics History

Antoine Barbez*, Foutse Khomh*, Yann-Gaël Guéhéneuc†
*Polytechnique Montreal, Canada; antoine.barbez@polymtl.ca, foutse.khomh@polymtl.ca
†Concordia University, Canada; yann-gael.gueheneuc@concordia.ca

Abstract—**Anti-patterns are poor solutions to recurring design problems. Number of empirical studies have highlighted the negative impact of anti-patterns on software maintenance which motivated the development of various detection techniques. Most of these approaches rely on structural metrics of software systems to identify affected components while others exploit historical information by analyzing co-changes occurring between code components. By relying solely on one aspect of software systems (i.e., structural or historical), existing approaches miss some precious information which limits their performances.**

In this paper, we propose CAME (Convolutional Analysis of code Metrics Evolution), a deep-learning based approach that relies on both structural and historical information to detect anti-patterns. Our approach exploits historical values of structural code metrics mined from version control systems and uses a Convolutional Neural Network classifier to infer the presence of anti-patterns from this information. We experiment our approach for the widely know God Class anti-pattern and evaluate its performances on three software systems. With the results of our study, we show that: (1) using historical values of source code metrics allows to increase the precision; (2) CAME outperforms existing static machine-learning classifiers; and (3) CAME outperforms existing detection tools.

Index Terms—**Anti-patterns, Deep Learning, Mining Software Repositories**

I. Introduction

During its development and maintenance, a software system may experience what Cunningham [6] called the *technical debt*, i.e., immature code and design choices implemented with the aim of meeting a deadline. The technical debt often takes the form of *anti-patterns*, i.e., poor solutions to recurring design problems, originally proposed by Fowler [13] and Brown et al. [4] along with refactoring operations aimed at removing them. There exists a variety of anti-patterns, which have been shown to negatively impact the maintainability [39] and comprehensibility [1] of the code. For example, the God Class anti-pattern, which happens when a class grows rapidly with the addition of new functionalities, violates the principle of single responsibility, which results in a class with low cohesion and high coupling.

A variety of approaches have been proposed to detect the occurrences of anti-patterns in source code [27], [34], [25]. Most of them rely on the formal definitions of anti-patterns and attempt to identify their occurrences in source code using structural metrics (e.g., Lines Of Code) along with empirically defined thresholds. However, anti-patterns can also be detected by an analysis of change history information [31], [29]. Indeed, the presence of anti-patterns in a system influence how source code entities evolve with one another over time. For example, the Feature Envy anti-pattern, which happens when a method is implemented in the wrong class, can be detected by identifying methods that change more often with methods of another class than those of their own class.

Although structural and historical anti-patterns detection techniques have shown acceptable performances, they are still far from perfect; they often report large numbers of false positives and false negatives. Also, we observe a complementarity in their detection results, meaning that some occurrences can not be detected by approaches based solely on structural or historical information [31]. Thus, existing approaches miss some precious complementary information which limits their performances. On the one hand, structural detection techniques rely on one single version of software systems. On the other hand, the historical detection technique does not consider the structural properties of the changed entities, nor the nature of the changes they have undergone.

Based on such considerations, we propose CAME (**C**onvolutional **A**nalysis of code **M**etrics **E**volution), a deep-learning based approach to detect anti-patterns by analyzing how source code metrics evolve over time when changes are applied to the system. Hence, the main idea behind our approach is to exploit the ability of deep-neural networks to identify key features in raw data with the aim of detecting anti-patterns from both structural and historical information. Concretely, structural metrics values related to the code components to be classified are computed at each revision of the system under investigation by mining its version control system (e.g., Git, SVN). This information is then organized into a two dimensional vector and fed through a Convolutional Neural Network (CNN) architecture to perform classification.

To the best of our knowledge, we are the first to exploit structural and historical information simultaneously for detecting anti-patterns.

This paper experiments the proposed approach for the detection of God Class. To train and assess the performances of our model, as well as to compare it with competing approaches, we used a manually-produced oracle containing occurrences of the studied anti-pattern in eight open-source java projects. To verify that our approach leverages historical information about code metrics to increase its performances, we firstly carried out an experiment in which we compare the performances of our model with different lengths of history (i.e., number of revisions). This, study answers the following research question:

(RQ1) To what extent historical values of source code metrics can improve detection performances?
Our results indicate that for God Class detection, the performances of our model significantly increase with the length of the input metrics history. With a metrics history of 500 revisions, the F-measure achieved by our model on the three test systems increases by $33\% \pm 6\%$ with respect to the performances achieved using one single revision.

Afterward, we compare the performances of CAME with those achieved by several *static* Machine Learning (ML) classifiers. Here and in the remainder of this paper, the term *static* refers to approaches that do not exploit historical information and therefore rely only on the current revision of the considered system, i.e., the revision in which we want to detect anti-patterns. Thus, we answer the following research question:

(RQ2) How does CAME compare to other static ML algorithms?
For God Class detection, our results show an overall improvement of 38% in term of F-measure, with respect to the classifier that achieved the best performances.

Finally, we evaluate our approach as a potential alternative to existing tools for helping practitioners to identify affected components to be refactored. To do so, we compare the performances of CAME with those of three state-of-the-art detection techniques: two static code analysis techniques and one approach that exploits change history information. With the results of this experiment, we aim to answer our last research question:

(RQ3) How does CAME compare to existing detection techniques?
CAME significantly outperforms existing approaches in detecting the God Class anti-pattern with an F-measure of 0.77. We show that it improves the precision by 196% and the recall by 51% with respect to the best competing technique. This suggests that our approach should be considered by practitioners for their software maintenance tasks.

The remainder of this paper is organized as follows. Section II provides background and discusses the related work. Section III describes our approach CAME. Section IV describes our case study design aiming to evaluate our approach for God Class detection. Section V reports the results of our first study aiming to assess the impact of the history length on CAME's performances, thus answering our first research question. Section VI reports the results obtained while comparing our approach with other techniques and answers the last two research questions. Section VII discusses the threats that could affect the validity of our results. Finally, Section VIII concludes with future work.

II. Background and Related Work

A. Definition of God Class

The God Class anti-pattern refers to the situation in which a class grows rapidly with the addition of new functionalities. A God Class implements a high number of responsibilities, delegating only trivial operations and accessing the data of many other classes. Consequently, this anti-pattern violates the design principle of uniform distribution of the system's intelligence among top-level classes [32] and has been shown to decrease program comprehension [1] and reusability.

B. Anti-patterns detection

The negative impact of anti-patterns on software quality highlighted by number of empirical studies [7], [38], [39], [1] has motivated the development of various automatic detection approaches. Most of these approaches attempt to identify bad motifs in models of source code using manually-defined metric-based heuristics.

First, Marinescu [24] proposed a mechanism for analyzing source code models with the aim of detecting design defects. This mechanism relies on a set of metrics (along with empirically defined thresholds) logically combined using AND/OR operators. Originally implemented for God Class, this approach called *detection strategy* has later been extended to 11 anti-patterns by Lanza and Marinescu [20] and implemented inside Eclipse plug-ins such as *InCode* [25]. Similarly, Moha et al. [27] proposed DECOR (DEtection and CORrection of Design Flaws). DECOR provides detection algorithms for four design anti-patterns (God Class, Functional Decomposition, Spaghetti Code, and Swiss Army Knife) and their 15 underlying code smells. This approach relies on so-called "Rule Cards" that encode the formal definitions of anti-patterns and code smells.

Anti-patterns are often defined along with refactoring operations designed to remove them, e.g., a God Class can be removed using the Extract Class Refactoring which consists in splitting the considered class into several more cohesive smaller classes. Consequently, occurrences of an anti-pattern can be detected by identifying opportunities to apply its corresponding refactoring operation. Based on such consideration, Fokaefs et al. [11] proposed an approach to detect God Classes in a system by suggesting a set of Extract Class Refactoring operations. Similarly, Tsantalis and Chatzigeorgiou [34] proposed an approach for automatic suggestions of Move Method Refactoring, i.e., methods that can potentially be moved to another class are considered as potential Feature Envy methods. These heuristics are implemented in the Eclipse plug-in *JDeodorant* [9], [10].

Anti-patterns also impact how code components evolve with one another over time, when changes are applied to the system. Consequently, Palomba et al. [29], [31] proposed HIST (Historical Information for Smell deTection), an approach to detect anti-patterns by analyzing co-changes occurring between code components. They experimented HIST on eight software systems, for the detection of five anti-patterns: Divergent Change, Shotgun Surgery, Parallel Inheritance, God Class and Feature Envy. They show that HIST is able to identify code smells that cannot be identified through approaches solely based on code analysis and suggest that better performances can be achieved by combining historical and structural information.

C. *ML-based Anti-patterns detection*

A number of approaches have used ML algorithms to detect anti-patterns. These models also rely on software metrics computed for each component to be classified. First, Kreimer [19] relied on decision trees to detect God Class and Long Method. This approach has later been extended to 12 anti-patterns by Amorim et al. [2]. Khomh et al. [17], [18] presented BDTEX (Bayesian Detection Expert), a metric based approach to build Bayesian Belief Networks from the definitions of anti-patterns. This approach has been experimented for the detection of God Class, Functional Decomposition, and Spaghetti Code. Maiga et al. [22], [23] proposed the use of Support Vector Machines to detect four well known anti-patterns: God Class, Functional Decomposition, Spaghetti code, and Swiss Army Knife.

Fontana et al. [12] conducted a large-scale study on the effectiveness of machine learning algorithms for anti-patterns detection. They experimented 16 different machine learning algorithms along with boosting techniques for the detection of four anti-patterns (Data Class, God Class, Feature Envy, and Long Method) on 74 software systems belonging to the `Qualitas Corpus` dataset [33]. They conclude that the application of machine learning to the detection of anti-patterns can provide high accuracy, even with a limited number of training examples.

More recently, Liu et al. [21] proposed a deep learning based approach to detect Feature Envy. Their model is a CNN fed with structural metrics as well as lexical information. They evaluated their approach on seven open-source applications. To train their model, they also propose an automatic approach for generating Feature Envy examples.

D. *Anti-patterns Through Systems' History*

Olbrich et al. [28] studied the impact of anti-patterns on the change behavior of code components. Specifically, they analyzed the historical data of two large scale software systems and compared the change frequency and size of components affected by God Class and Shotgun Surgery with those of healthy components. Vaucher et al. [36] studied the "life cycle" of God Class occurrences in two open-source systems with the aim of understanding when they are created and how they evolve. They used a Bayesian Belief Network to track the evolution of the"godliness" of these classes through different versions of the studied systems. Similarly, Chatzigeorgiou and Manakos [5] tracked the evolution of three anti-patterns: Long Method, Feature Envy and State Checking in the history of two open-source systems. Finally, Tufano et al. [35] performed the largest experiment on the presence of anti-patterns through the history of software systems. Specifically, they mined the history of 200 software projects to understand under what circumstances anti-patterns appear. First, their results confirm the observation made by Chatzigeorgiou and Manakos [5] that most of instances are introduced when the file is added to the system. Second, they show that anti-patterns are also often introduced the last month before deadlines by experienced developers.

III. CAME: CONVOLUTIONAL ANALYSIS OF CODE METRICS EVOLUTION

This section presents CAME, our deep-learning based approach to detect anti-patterns from source code metrics historical information. We first describe the input of our model before describing its architecture. Finally, we discuss the procedure we followed to train this architecture for anti-patterns detection.

A. *Input*

To detect instances of a given anti-pattern in a system, our approach uses a deep-learning based classifier to perform a boolean prediction on each code component (i.e., class or method) of the system. To perform this prediction, we exploit both structural and historical information. To define the input of our model, we first select a set of N_m structural metrics which can be computed for any code component of the system under investigation. Then, we walk through the history of revisions of the system in reverse order by mining its repository. At each revision, we recalculate the values of the selected metrics for each component. We refer to the value of the j^{th} metric computed for a component c at the i^{th} revision of the system, as $m_{i,j}(c) \in \mathbb{R}$. Thus, for a given code component c to be classified, the input of our model is a real valued matrix $\mathbf{X}_c$ such as:

$$\mathbf{X}_c(i,j) = m_{i,j}(c) \tag{1}$$

Different software systems experience a different number of revisions (i.e., commits). Also, a code component may have been introduced in a system during its creation or on the contrary during a recent revision. As a consequence, different code components may be characterized by a different number of revisions. To allow our model to receive a fixed size input, we limit the length of the metrics history (i.e., number of revisions considered for a given component) to a constant L_h, which is an hyper-parameter to be adjusted. Hence, each input matrix of our model is of shape: $L_h \times N_m$. If a component has an history shorter than L_h, meaning that the component did not exist in the early revisions of the system, we pad the rest of its input matrix with zeros.

B. *Architecture*

Figure 1 overviews the architecture of the convolutional neural network used by our approach to perform classification. The model contains several convolution + pooling layers fully-connected to a Multi Layer Perceptron (MLP) model, i.e., several dense layers connected to an output layer. The convolution layers perform 1D convolutions with a *stride* of *1* and a *tanh* activation function. The output of each convolution layers is fed to max-pooling layers that reduce the dimentionality

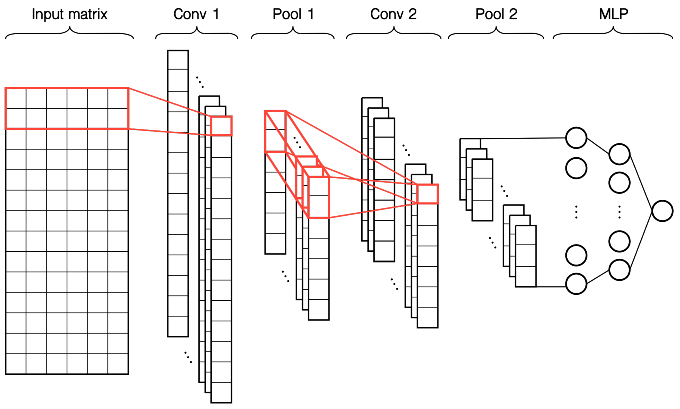

Fig. 1. Architecture of the CNN model used by CAME. In this example, the network contains two convolution + polling layers with filters of size 2 and two dense hidden layers.

of their input by taking the maximum value across a small spatial region. Then, after being flattened, the output of the last pooling layer is fed to *tanh* dense layers. Finally, the output layer is made of one single *sigmoid* neuron which outputs the predicted probability that a component is affected given its input matrix.

The choice of such architecture has been motivated by the ability of CNNs to extract high level features from high dimensional data, e.g., deep CNNs used in image processing recognize complex shapes in raw pixels. Indeed, our conjecture is that the first convolution layer can detect that some metrics have changed between two consecutive commits. Then, the next layers build a representation of the changes that characterize an anti-pattern. Therefore, our model identify recurrent patterns in the local variations of software metrics, which we believe constitute a useful complementary information for detecting anti-patterns.

C. Training

This subsection presents the consideration we adopted to train our model but we assume that the following applies to anyone willing to train neural-networks on the task of anti-patterns detection. First, let $\mathcal{D} = \{(\mathbf{X}_i, y_i)\}_{i=1}^{n}$ be our training set. With $\mathbf{X}_i \in \mathbb{R}^{L_h \times N_m}$, the input matrix (cf. equation 1) of the i^{th} component to classify, $y_i \in \{0, 1\}$ the true label for this component and n the number of instances in the training set.

Also, we refer to the set of weights of our model as $\boldsymbol{\theta} = \{\mathbf{w}_l\}_{l=1}^{L}$, with $\mathbf{w}_l$ being the weight matrix of the l^{th} layer and L the number of layers in the network. Finally, we note $P_{\boldsymbol{\theta}}(1|\mathbf{X}_i) = \mathrm{g}(\mathbf{X}_i.\boldsymbol{\theta})$ the predicted probability outputed by our model for the i^{th} component, with $\mathrm{g}(x) = \frac{1}{1+e^{-x}}$ the *sigmoid* function.

1) Loss Function: In a software system, components affected by an anti-pattern are usually a minority ($\approx 1\%$) [30]. Classifiers optimized using conventional loss functions, e.g., *cross entropy*, on such data tend to favor the majority category, thus maximizing the overall accuracy [14]. This characteristic known as the "imbalanced data problem" prevents us from using such loss function to guide the optimization of our model. Hence, we must define a loss function that maximizes our evaluation metric: the F-measure expressed in equation 9.

Optimizing a model through gradient decent requires computing the gradient of the loss with respect to the model weights. However, computing the number of true positives TP and positives m_{pos} (cf. Table II) requires counting elements from the probability outputed by the model, which necessarily involves discontinuous operators:

$$TP(\boldsymbol{\theta}, \mathcal{D}) = \sum_{\substack{i=1 \\ y_i=+1}}^{n} \delta(P_{\boldsymbol{\theta}}(1|\mathbf{X}_i) > 0.5) \tag{2}$$

$$m_{pos}(\boldsymbol{\theta}, \mathcal{D}) = \sum_{i=1}^{n} \delta(P_{\boldsymbol{\theta}}(1|\mathbf{X}_i) > 0.5) \tag{3}$$

With:

$$\delta(x) = \begin{cases} 1 \text{ if } x\text{=True} \\ 0 \text{ if } x\text{=False} \end{cases}$$

This characteristic prevents us from using the F-measure directly to define our loss function. Consequently, we use the continuous and differentiable approximation of the F-measure provided by Jansche [16] which consists in considering the limit:

$$\delta(P_{\boldsymbol{\theta}}(1|\mathbf{x}_i) > 0.5) = \lim_{\gamma \to +\infty} \mathrm{g}(\gamma \mathbf{X}_i.\boldsymbol{\theta}) \tag{4}$$

Hence, we define our loss function as $loss = -\tilde{F}_m(\boldsymbol{\theta}, \mathcal{D})$ with:

$$\tilde{F}_m(\boldsymbol{\theta}, \mathcal{D}) = 2 \times \frac{\sum_{\substack{i=1 \\ y_i=+1}}^{n} \mathrm{g}(\gamma \mathbf{X}_i.\boldsymbol{\theta})}{n_{pos} + \sum_{i=1}^{n} \mathrm{g}(\gamma \mathbf{X}_i.\boldsymbol{\theta})} \tag{5}$$

Note that the value of the hyper-parameter γ will be adjusted along with other hyper-parameters during the tuning of our model.

2) Regularization: Overfitting occurs when a statistical model fails to generalize to new examples by learning irrelevant characteristics of its training data. Hence, an overfitted model performs well on the training set but achieves poor performances on unseen test data. To prevent our model from overfitting, we used the widely adopted L_2 regularization technique, which consists in adding a term to the loss function to encourage the weights to be small [37]. This term rely on the Euclidean norm of the weight matrices, i.e., $\|\mathbf{w}\|_2 = \sqrt{\mathbf{w}^\top \mathbf{w}}$, also called L_2-norm. Thus, the L_2 regularization term added to the loss function can be expressed as:

$$L_2 = \lambda \sum_{l=1}^{L+1} \|\mathbf{w}_l\|_2 \tag{6}$$

With $\lambda \in \mathbb{R}$ an hyper-parameter adjusted during cross-validation.

IV. Study Design for God Class

In this section, we lay the foundations of our study aiming to evaluate the effectiveness of our approach CAME for detecting the God Class anti-pattern. After describing the software systems investigated in this work, we present the metrics selected for God Class and overview the process followed to extract historical information about these metrics. We finally describe our evaluation approach and replication package. Our study aims at addressing the following three research questions:

(RQ1) To what extent historical values of source code metrics can improve detection performances?
This research question assesses the impact of the length of the input metrics history (i.e., L_h) on our models' performances. Hence, before comparing our approach with other detection techniques, we can confirm that metrics history provides relevant information to our model and establish which value of L_h leads to optimal performances.

(RQ2) How does CAME compare to other static ML algorithms?
This research question aims at comparing the performances of our approach with other static ML algorithms, i.e., which rely on one single revision of the systems.

(RQ3) How does CAME compare to existing detection techniques?
This research question aims at comparing our approach with existing detection techniques. The results of this comparison will provide insights on the advantages of using CAME instead of other detection tools to help developers in their daily maintenance tasks.

A. Studied Systems

To answer our research questions, we base our study on eight open-source Java projects belonging to various ecosystems. Android Opt Telephony and Android Support belong to the Android APIs[1]. Apache Ant, Apache Tomcat, Apache Lucene, and Apache Xerces belong to the Apache Foundation[2]. ArgoUML[3] is a software design tool and Jedit[4] a text editor. For the sake of simplicity, we chose to analyze only the directories that implement the core features of the systems and to ignore test directories.

The choice of these systems has been motivated by the fact that they have been used for a similar purpose in prior studies. Indeed, to train our model and compare its performances with those of other approaches, we needed an oracle reporting the occurrences of God Classes in a set of software systems. Unfortunately, we found no such large dataset in the literature. Consequently, we reused the occurrences of God Class used to evaluate the approaches HIST [31] and DECOR [27], made publicly available[5][6] by their respective authors. Among the systems available, we kept only those for which the full history was available through Git or SVN. Also, for some systems, we found occurrences that do not belong to it or that do not exist in the current revision. In such cases, we did not incorporate the systems in our oracle. Table I overviews the main characteristics of the subject systems.

TABLE I
Characteristics of the Studied Systems

System name	Snapshot	Directory	#Files	#GCs
Android Opt Telephony	c241cad	src/java/	190	10
Android Support	38fc0cf	v4/	104	4
Apache Ant	e7734de	src/main/	755	7
Apache Lucene	39f6dc1	src/java/	160	3
Apache Tomcat	398ca7ee	java/org/	1005	5
Apache Xerces	c986230	src/	658	15
ArgoUML	6edc166	src_new/	1246	22
Jedit	e343491	./	437	5

B. Source Code Metrics

For God Class detection, the components to classify correspond to the classes of the system. However, we chose to consider only top-level-classes as potential God Classes, as we found no inner-classes positively labeled in our data. To decide whether or not a given class is affected by the God Class anti-pattern, we retrieve the history of **seven** structural metrics. Note that to compute these metrics, we do not consider attributes and methods of inner- (or nested-) classes as components of the class under investigation. Indeed, the refactoring operation commonly applied to remove God Classes (Extract Class Refactoring) consists in identifying one or several group of attributes and methods of the class dedicated to one functionality and then extract them as a separate class. Thus, we assume that inner-classes may be the result of such refactoring. In the following, we define each selected metric and provide a brief rational for their use.

- **ATFD** (Access To Foreign Data): Number of distinct attributes of unrelated classes (i.e., not inner- or super-classes) accessed (directly or via accessor methods) in the body of a class. A God Class accesses a lot of data from other classes as suggested by Lanza and Marinescu [20].
- **LCOM5** (Lack of COhesion in Methods): Measures cohesion among methods of a class based on the attributes accessed by each method [15]. A God Class handles many unrelated functionalities, thus methods related to different functionalities access different sets of attributes. This metrics is also part of the detection process of DECOR [27].
- **LOC** (Lines Of Code): Sum of the number of lines of code of all methods of a class. A God Class implements a high number of functionalities, thus it is mainly characterized by its size.

[1]https://android.googlesource.com/
[2]https://www.apache.org/
[3]http://argouml.tigris.org/
[4]http://www.jedit.org/

[5]http://www.rcost.unisannio.it/mdipenta/papers/ase2013/
[6]http://www.ptidej.net/tools/designsmells/materials/

- **NAD** (Number of Attributes Declared): Number of attributes declared in the body of a class. This metrics is part of the detection process of DECOR [27].
- **NADC** (Number of Associated Data Classes): Number of dependencies with data-classes (i.e., data holders without complex functionality other that providing access to their data). This metrics is part of the detection process of DECOR [27].
- **NMD** (Number of Methods Declared): Number of non-constructor and non-accessor methods declared in the body of a class. This metrics is part of the detection process of DECOR [27].
- **WMC** (Weighted Method Count): Sum of the McCabe's cyclomatic complexity [26] of all methods of a class. A God Class has a high functional complexity, as suggested by Lanza and Marinescu [20].

C. Data Extraction

As previously evoked, we construct the input matrices of our model by mining software repositories and navigating through the different revisions of a system using its version control API (e.g., Git). To automate this process and allow replication, extension and reuse of our work, we designed a component called `RepositoryMiner`[7] which automatically extracts the source code metrics history of any java software system. The `RepositoryMiner` currently implements 12 class- and method-related structural metrics and allows to easily define new ones. We briefly describe each step of the process followed to extract our data below.

Input: The `RepositoryMiner` takes as input three arguments: (1) the URL of the system's repository; (2) the SHA, i.e., the identification number, of the system's snapshot (i.e., commit) we want to analyze and; (3) the sub-directories of interest, i.e., those in which we want to detect affected components.

Initialization After downloading the system repository, we checkout to the current snapshot and parse the Abstract Syntax Tree (AST) of all the *.java* files contained in the sub-directories of interest. This step creates a `SystemObject` that holds a representation of the system's classes which will be used later to compute our metrics.

Data Extraction We walk through all the commits of the system in reverse order starting from the current snapshot. At each commit, we perform the following steps:

1) Retrieve the names of all the files that have been changed between the current and the previous revision.
2) If the changed files belong to the `SystemObject`, checkout and update the corresponding classes.
3) Check if any component (i.e., file, class, or method) have been renamed between the current and the next revision.
4) Compute the code metrics values for each class of the `SystemObject` by taking into account renamed components.

[7]https://github.com/antoineBarbez/RepositoryMiner/

Output The `RepositoryMiner` outputs a collection of *.csv* metric files. Each file contains the code metrics values computed for each class of the system at a given revision. Note that before creating a new metric file, we check if at least one value has been modified with respect to the previous one.

D. Evaluation

To evaluate the performances of CAME as well as those of the competing classifiers, we selected three systems, i.e., Android Support, Apache Tomcat, and Jedit, among the eight software systems presented in Table I. These systems have been selected for the sake of generalizability. Indeed, they belong to different domains: telephony framework, service container, and text editor and have different sizes (i.e., number of classes). The remaining five systems are used for training and tuning the hyper-parameters of the different ML-based approaches investigated in this work.

We compute the overall performances of each approach by running it on all instances (i.e., the java classes) of the three test systems. Indeed, each classifier is able to perform a boolean prediction on each single instance. Then, we evaluate a classifier using the so-produced confusion matrix presented in Table II.

TABLE II
CONFUSION MATRIX FOR BINARY CLASSIFICATION

		predicted label		***total***
		1	0	
true label	1	TP	FN	n_{pos}
	0	FP	TN	n_{neg}
total		m_{pos}	m_{neg}	n

With TP the number of true positives; FN the number of false negatives (i.e., misses); FP the number of false positives and TN the number of true negatives. We use this matrix to compute our evaluation metrics:

$$precision = \frac{TP}{TP + FP} \quad (7) \qquad recall = \frac{TP}{TP + FN} \quad (8)$$

In order to evaluate each approach with a single aggregated metric, we also compute the F-measure (i.e., the harmonic mean of precision and recall):

$$F_m = 2 \times \frac{precision \times recall}{precision + recall} = 2 \times \frac{TP}{n_{pos} + m_{pos}} \quad (9)$$

E. Replication Package

To facilitate further evaluation and reuse of our work, all the data used in the context of this study, as well as our implementation of CAME for God Class is publicly available in our online appendix[8].

[8]https://github.com/antoineBarbez/CAME/

TABLE III
CAME Hyper-parameters Tuning

Hyper-parameter	Range
Learning Rate (η)	$10^{-[0.0;2.5]}$
L2-norm (λ)	$10^{-[0.0;2.5]}$
Gamma (γ)	$[1;10]$
# Conv Layers	$[0,1]$ if $L_h \leq 10$ else $[1;2]$ if $L_h \leq 100$ else 2
# Filters	$[10;60]$
Filter size	$[2;4]$
Pool size	$\{2,5,10\}$ if $L_h \leq 100$ else $\{5,10,15,20\}$
# Dense Layers	$[1;3]$
Dense Layer size	$[4;100]$ then $[4;s]$

With s the size of the previous dense layer.

V. Assessing the Impact of the Metrics History Length on CAME's Performances

In this section, we report the results of our experiments conducted with the aim of assessing the impact of the metrics history length (L_h) on the performances of our approach. Hence, this section answers the first research question.

A. Approach

To answer **RQ1**, we monitor the performances achieved by CAME, in terms of precision, recall and F-measure, with different lengths of metrics history: $L_h \in \{1, 10, 50, 100, 250, 500, 1000\}$. For each value of L_h experimented, we build and train 10 distinct CNNs in order to retrieve the mean and standard deviation of the three performance metrics achieved for each length. To avoid any bias in our conclusions, we must consider that the length of the metrics history may affect the optimal values of the other hyper-parameters of our model. Consequently, before training our model for a new value of L_h, we perform a new tuning of its hyper-parameters. As explained in Section IV-D the performances metrics are computed by considering instances of the three test systems together: Apache Tomcat, Jedit and Android Support. The remaining five systems are kept for training and hyper-parameters tuning.

B. Hyper-parameters Tuning

We select the optimal set of hyper-parameters of our model for each history length investigated using a random search over 100 generations of nine hyper-parameters [3]. Table III reports for each hyper-parameter, the range of values experimented. We monitor the performances achieved by our model with different sets of hyper-parameters by carrying out a 5-fold cross-validation. Thus, we first split the training set into 5 equal size partitions, i.e., folds. Then, at each iteration, we generate a new random set of hyper-parameters and compute our model's prediction on each fold by leaving it out while keeping the others for training (100 epochs). Finally, we concatenate the obtained predictions and compute the overall F-measure. The optimal values retained after tuning for each history length can be found in our online appendix[9].

[9]https://github.com/antoineBarbez/CAME/tree/master/experiments/tuning/

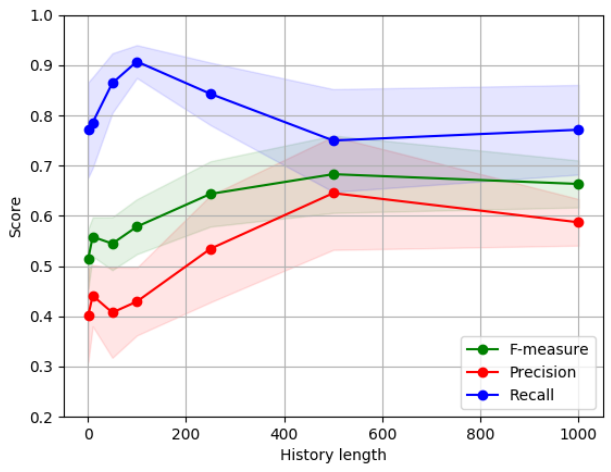

Fig. 2. Comparison of CAME's performances for different sizes of metrics history. The lines show mean values while areas show standard deviations over 10 trainings.

For each history length investigated, we train our model using the previously found hyper-parameter values. We perform a mini-batch based stochastic gradient descent optimization during 300 epochs, with 5 mini-batches and an exponential learning rate decay of .5 every 100 epochs. It is important to remember that we train 10 randomly initialized CNNs per history length investigated in order to compute the mean and standard deviation of their performance metrics.

C. Results

Figure 2 presents the results of our comparison for God Class, i.e., mean values and standard deviations of CAME's performance metrics over the three test systems, obtained using different sizes of metrics history (1, 10, 50, 100, 250, 500, 1000).

RQ1: To what extent historical values of source code metrics can improve detection performances?

For God Class detection, our results show that in term of F-measure, the overall performances achieved by CAME on the three test systems significantly increase with the size of the metrics history. Specifically, after training our model ten times per history length, we observe that the overall F-measure improves from 0.51 ± 0.06 for $L_h = 1$ to 0.68 ± 0.08 for $L_h = 500$ (improvement of $33\% \pm 6\%$). Regarding the other two performance metrics, we see that although CAME achieves a better recall on average for $L_h > 1$, there does not seem to be any real correlation between recall and history length. As shown in Figure 2 the recall strongly increases in the range $[1; 100]$ but then drops to recover its initial value at $L_h = 1000$. Finally, we can see that the precision clearly improves with the history length similarly to the F-measure. By comparing the precision for $L_h = 1$ with respect to the history length that led to the best results ($L_h = 500$), we observe an overall improvement of $61\% \pm 11\%$.

Also, we can see that the performances of CAME (especially precision) decrease for $L_h > 500$. This observation is surprising considering that a longer metrics history includes the shorter ones and thus should lead to a greater or equal F-measure. We believe this could be due to a sub-optimal choice of the hyper-parameters. Indeed, the randomness of the process and the finite number of combinations experimented make it hard to conclude that the hyper-parameters selected are optimal. Furthermore, while tuning our model, we limited the number of convolution layers to 2. Hence, it is possible that more convolution layers may be needed to process a history of size $L_h = 1000$ and achieve optimal performances.

Our results confirm that our model properly leverages historical information about source code metrics by decreasing the number of false positives. Specifically, for $L_h > 1$ we observe a recall greater or equal to that obtained with a single revision of the input metrics and more importantly, we see that the precision clearly increases with the length of the input metrics history.

VI. Comparing CAME with Other Approaches

This section reports the results of our experiments aiming to compare CAME with other approaches. To avoid redundancies, we report the results for our last two research questions together.

A. Approach

Similarly to the previous study, we evaluate the performances of the different approaches on our three test systems (i.e., Apache Tomcat, Jedit, and Android Support) while keeping the other five for training and hyper-parameters calibration. To compute the performances of CAME on these systems, we reuse the ten CNNs trained during our previous study (cf. Sections V-A) with $L_h = 500$. As shown in Figure 2, after being trained, each CNN achieves different performances due to the randomness of its initialization. Hence, the final performances of CAME are computed from the outputs of the ten CNNs using an ensemble method, also known as *boosting technique*. Such method have been shown to lead to better performances than those of each independent classifier [8]. In the context of this study, we use the widely-adopted Bayesian averaging heuristic to compute the ensemble prediction. Thus, the final predicted probability that a class c is a God Class can be expressed as:

$$P_{ensemble}(1|\mathbf{X}_c) = \frac{\sum_{i=1}^{10} P_i(1|\mathbf{X}_c)}{10} \tag{10}$$

with $P_i(1|\mathbf{X}_c)$, the predicted conditional probability given by the i^{th} CNN and $\mathbf{X}_c$ the input matrix of the class c.

To answer **RQ2**, we compare CAME with three ML classifiers: Decision Tree, Multi Layers Perceptron (MLP), and Support Vector Machine (SVM). The input of these classifiers consists in the same seven metrics used by CAME (cf. Section IV-B) computed on the current revision of each studied system. Similarly to CAME, we first calibrate the hyper-parameters of each classifier and compute the final performances using the Bayesian averaging ensemble method on ten pre-trained models.

TABLE IV
Hyper-parameters Tuning of the Competing ML Classifiers

Model	Hyper-parameter	Range
Decision Tree	Max Features	$\{sqrt, log2, None\}$
	Max Depth	$10 \times [1; 10]$
	Min Sample Leaf	$\{1, 2, 4, 6\}$
	Min Sample Split	$\{2, 5, 10, 15\}$
MLP	Learning Rate (η)	$10^{-[0.0;2.5]}$
	L2-norm (λ)	$10^{-[0.0;2.5]}$
	Gamma (γ)	$[1; 10]$
	# Dense Layers	$[1; 3]$
	Dense Layer size	$[4; 100]$ then $[4; s]$
SVM	Penalty	$\{0.001, 0.01, 0.1, 1, 10, 100\}$
	Gamma	$\{0.001, 0.01, 0.1, 1, 10, 100\}$
	Kernel	$\{linear, rbf, sigmoid\}$

With s the size of the previous dense layer.

To answer **RQ3**, we compare CAME with three state-of-the-art detection approaches. Two static code analysis techniques: DECOR [27] and JDeodorant [10] and one approach that exploits change history information to detect anti-patterns: HIST [31]. We chose to compare CAME with these approaches to increase the scope of our study. Indeed, they rely on radically different strategies to detect anti-patterns and are thus likely to be complementary. DECOR rely on the use of *Rule Cards* that encode the formal definitions of anti-patterns using structural and lexical information. JDeodorant detects affected components by identifying refactoring opportunities. Finally, HIST exploits change history information derived from version control systems. To compute the performances of each approach, we used the implementations made publicly-available by their respective authors whenever possible and replicated the approaches for which no implementation was available. Thus, we ran DECOR using the Ptidej API[10] and JDeodorant using its Eclipse plug-in[11]. For HIST, we implemented the detection rules as described in its original paper [31]. Also, we implemented our own component[12] to extract code changes at a class level granularity because the original component was not available due to its license.

B. Hyper-parameters Tuning

We calibrate the hyper-parameters of each ML algorithm using the same procedure adopted for CAME in the previous study (see Section V-B). Hence, we use a random search of 100 iterations over a variety of hyper-parameters related to each classifier. Also, to monitor the performances induced by each set of hyper-parameters, we use a 5-fold cross-validation with instances of the five training systems. Table IV reports for each classifier, the set of hyper-parameters tuned, as well as the ranges of values experimented. To train the MLP model, we used rigorously the same procedure followed for training CAME.

[10]https://github.com/ptidejteam/v5.2/
[11]https://marketplace.eclipse.org/content/jdeodorant/
[12]https://github.com/antoineBarbez/HistoryExtractor

TABLE V
PERFORMANCES FOR GOD CLASS DETECTION

Approaches	Apache Tomcat			JEdit			Android Platform Support			**Overall**		
	Precision	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*
DECOR	68%	40%	50%	17%	60%	26%	–	0%	–	24%	36%	29%
HIST	–	0%	–	25%	40%	31%	18%	100%	31%	20%	43%	27%
JDeodorant	2%	60%	5%	5%	60%	9%	50%	50%	50%	4%	57%	8%
Decision Tree	33%	20%	25%	100%	40%	57%	100%	25%	40%	68%	29%	40%
MLP	25%	100%	40%	68%	80%	73%	100%	75%	86%	41%	**86%**	56%
SVM	50%	20%	29%	100%	20%	33%	–	0%	–	68%	14%	24%
CAME	50%	100%	68%	100%	80%	89%	100%	75%	86%	**71%**	**86%**	**77%**

C. Results

Table V reports the performances for God Class detection, in terms of precision, recall and F-measure, achieved by CAME along with those of the competing techniques. The performances are reported for each subject system and on overall, i.e., considering the three test systems as a single one. When an approach did not detect any occurrence of God Class in a system, it was possible to compute neither the precision, nor the F-measure. In these cases a "–" is indicated in the corresponding cell.

RQ2: How does CAME compare to other static ML algorithms?

For God Class detection, our results show that on overall, our approach significantly outperforms the three classifiers. In term of F-measure the performances improve from 56% to 77% (improvement of 38%) with respect to the classifier with the best performances (the MLP). Unsurprisingly, CAME achieves the same recall than the MLP. Indeed, it is interesting to remark that a MLP model is equivalent to the CNN used by CAME with no convolution layers and considering $L_h = 1$. Hence, we make the same observation than in the previous study regarding recall. Considering the other two classifiers we clearly see that they can not compete with CAME in term of recall with 29% for the Decision Tree algorithm and 14% for the SVM against 86% for our approach. Finally, CAME ensured on overall a better precision than any other classifier (+ 18% on average). Regarding the performances achieved on each system, we see that CAME achieves the highest precision (100%) on two of the three test systems but only 50% on Apache Tomcat. This may be due to the fact that this system is the largest of our training set but contains only five occurrences of God Class. Hence, any model is more likely to have a low precision on it. Finally, our approach seem to have a more stable recall with values ranging between 75% and 100%.

CAME significantly outperforms other static ML classifiers. Indeed, none of the competing algorithms performs better than our approach on any system and considering any performance metric.

RQ3: How does CAME compare to existing detection techniques?

Our results show that CAME clearly outperforms existing detection methods in detecting the God Class anti-pattern. Indeed, CAME shows on overall, a precision of 71% and a recall of 86% (F-measure of 77%) over the test systems. With respect to the tool that performs the best for each performance metric, we see that CAME improves the precision by 196%, the recall by 51% and the F-measure by 166%. Also, as we can see, each tool achieves poor performances on at least one system, which is not the case for our approach. This confirms that CAME performs well independently of the systems characteristics.

However, some factors can explain the poor performances reported for some of the approaches experimented. First, as the original implementation of HIST is not publicly available, we had to replicate this approach from the directives given by the authors in the paper [31]. We are aware that some differences in our respective implementations may have impacted the performances reported. Second, it is important to note that JDeodorant is not, strictly speaking, an anti-pattern detection tool. Instead, JDeodorant suggests opportunities to apply refactoring operations in the system. Hence, it is not surprising that it achieves a high recall at the expense of its precision, because it may exists a high number of classes in the subject systems that could benefit from an Extract Class Refactoring operation without necessarily being God Classes.

CAME significantly outperforms all the existing techniques investigated in this work for God Class detection. This suggests that our approach should be considered by practitioners for identifying affected code components to be refactored.

VII. THREATS TO VALIDITY

In this section, we discuss the threats that could affect the validity of our results.

a) Construct Validity: Threats to construct validity concern the relation between theory and observation. This could be due to how our oracle was build (cf. Section IV-A). To combat this limitation, we examined the occurrences reported in respectively HIST and DECOR's replication packages before incorporating them in our oracle. Also, both papers have been awarded by the community which convince us of the reliability of their data. Also, the metrics we selected for God Class detection have been chosen on the basis of their use in literature. However, other metrics that we have overlooked may have led to better performances. Yet, our approach does not focus on any specific metric but on the potential improvements induced by the use of metrics history in general. Another threat is related to the implementations used to evaluate existing approaches. To replicate HIST, we followed rigorously the guidelines provided by the authors. However, some differences may remain between our respective implementations.

b) External Validity: Threats to external validity concern the generalizability of our findings. In the context of our study, this could refer to the number of software systems on which we experimented our approach. Manually-build datasets are time consuming to produce for anti-patterns which explains their rareness in literature. Hence, our sample size may limit the generalizability of our results and we agree that further evaluation of our approach on a larger set of systems would be desirable. To reduce this threat, the software systems used for evaluation have been selected for their different sizes and domains. This threat could also refer to the fact that we experimented our approach for the detection of one single anti-pattern: God Class. Yet, we assume that our approach could be applied to any anti-pattern and we plan to extend it in a future work.

c) Internal Validity: Threats to internal validity concern all the factors that could have impacted our results. This could refer to the fact that the ML classifiers investigated in this work, i.e., CAME, Decision Tree, MLP, SVM, are not deterministic approaches. As a consequence, the predictions made by these classifiers may vary depending on their initialization which makes them difficult to compare. To reduce this threat, we base our evaluation on ten independent models per classifier. Hence we consider their individual performances as a Gaussian distribution in our first study (cf. Section V) and use an ensemble method in our second study (cf. Section VI). Also, we calibrated the hyper-parameters of all the approaches investigated in this work whenever necessary. Finally, Another threat is related to the choice of the ML model used by our approach to process the input metrics history. As explained in Section III-B, we chose a CNN model because we believe that applying successive convolutions to the input allows our model to learn a representation of the changes that characterize an anti-pattern. However, this decision was only based on intuition and we cannot exclude that other deep-learning models (e.g., RNNs) would lead to better performances.

VIII. Conclusion and Future Work

The impact of anti-patterns on software quality highlighted by number of empirical studies has motivated the development of various detection techniques. Although the proposed approaches have helped developers in identifying affected code components to be refactored, we identified a major limitation common to these works. Different detection techniques rely on different sources of information and thus, identify different sets of occurrences of anti-patterns. Hence, by ignoring either structural or historical aspects of software systems, existing approaches miss some precious information which limits their performances. Consequently, we proposed CAME a deep-learning based approach that relies on both structural and historical information to detect anti-patterns. Our approach exploits historical values of structural code metrics and uses a CNN classifier to infer the presence of anti-patterns from this information. We implemented our approach for the detection of God Class and evaluated it on three software systems. With the results of our study, we showed that:

- The performances of CAME increase with the length of the metrics history fed through our model. This shows that our model properly leverages historical information to improve its performances.
- CAME significantly outperforms other ML-based classifiers that do not rely on historical data.
- CAME significantly outperforms existing detection tools.

Our short term research agenda includes extending our approach to other anti-patterns as well as further evaluation of CAME on a greater number of systems. We also plan to investigate the use of deep-learning visualization techniques [40] on the architecture of CAME. We believe that such approach could help us identifying the root causes and characteristics of anti-patterns.

References

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Software maintenance and reengineering (CSMR), 2011 15th European conference on*, pages 181–190. IEEE, 2011.

[2] Lucas Amorim, Evandro Costa, Nuno Antunes, Baldoino Fonseca, and Marcio Ribeiro. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 261–269. IEEE, 2015.

[3] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

[4] William H Brown, Raphael C Malveau, Hays W McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[5] Alexander Chatzigeorgiou and Anastasios Manakos. Investigating the evolution of bad smells in object-oriented code. In *International Conference on the Quality of Information and Communications Technology*, pages 106–115. IEEE, 2010.

[6] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993.

[7] Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129–143, 2004.

[8] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

[9] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 519–520. IEEE, 2007.

[10] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 1037–1039. IEEE, 2011.

[11] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, 2012.

[12] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191, 2016.

[13] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[14] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge & Data Engineering*, (9):1263–1284, 2008.

[15] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.

[16] Martin Jansche. Maximum expected f-measure training of logistic regression models. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 692–699. Association for Computational Linguistics, 2005.

[17] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. A bayesian approach for the detection of code and design smells. In *Quality Software, 2009. QSIC'09. 9th International Conference on*, pages 305–314. IEEE, 2009.

[18] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572, 2011.

[19] Jochen Kreimer. Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science*, 141(4):117–136, 2005.

[20] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[21] Hui Liu, Zhifeng Xu, and Yanzhen Zou. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 385–396. ACM, 2018.

[22] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabane, Yann-Gael Gueheneuc, and Esma Aimeur. Smurf: A svm-based incremental anti-pattern detection approach. In *Reverse engineering (WCRE), 2012 19th working conference on*, pages 466–475. IEEE, 2012.

[23] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. Support vector machines for anti-pattern detection. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pages 278–281. IEEE, 2012.

[24] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 2004.

[25] Radu Marinescu, George Ganea, and Ioana Verebi. Incode: Continuous quality assessment and improvement. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 274–275. IEEE, 2010.

[26] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[27] Naouel Moha, Yann-Gaël Guéhéneuc, Duchien Laurence, and Le Meur Anne-Franccoise. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering (TSE)*, 36(1):20–36, 2010.

[28] Steffen Olbrich, Daniela S Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 390–400. IEEE, 2009.

[29] F. Palomba, G. Bavota, M. Di Penta, R. Oliverto, D. Poshyvanyk, and A. De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2015.

[30] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221, 2018.

[31] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *ASE*, pages 268–278, 2013.

[32] Arthur J Riel. *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[33] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336–345. IEEE, 2010.

[34] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[35] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 403–414. IEEE Press, 2015.

[36] Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. Tracking design smells: Lessons from a study of god classes. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 145–154. IEEE, 2009.

[37] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[38] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 306–315. IEEE, 2012.

[39] Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 682–691. IEEE Press, 2013.

[40] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.