Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection

Hui Liu, *Member, IEEE*, Qiurong Liu, Zhendong Niu, and Yang Liu

Abstract—Most code smell detection tools expose thresholds to engineers for customization because code smell detection is essentially subjective and application specific. Another reason why engineers should customize these thresholds is that they have different working schedules and different requirements on software quality. They have their own unique need on precision and recall in smell detection. This unique need should be fulfilled by adjusting thresholds of smell detection tools. However, it is difficult for software engineers, especially inexperienced ones, to adjust often contradicting and related thresholds manually. One of the possible reasons is that engineers do not know the exact quantitative relation between threshold values and performance, e.g., precision. In this paper, we propose an approach to adapting thresholds automatically and dynamically. Engineers set a target precision manually according to their working schedules and quality requirements. With feedback from engineers, the proposed approach then automatically searches for a threshold setting to maximize recall while having precision close to the target precision. The proposed approach has been evaluated on open-source applications. Evaluation results suggest that the proposed approach is effective.

Index Terms—Software refactoring, code smells, feedback control, smell identification

1 Introduction

Software refactoring is an important activity to improve software quality by adjusting software internal structure [1], [2]. One of the key issues in software refactoring is to identify code smells that require refactoring [1]. However, manual identification of code smells from large applications is challenging. Thus, researchers have proposed a number of algorithms and tools to detect code smells automatically or semi-automatically. The performance of these detection tools, e.g., precision and recall, is critical for the success of software refactoring activities. However, Bowes et al. [3] observed that different code smell identification tools that are designed to identify the same kind of code smells might report different results. Besides the significant differences in the definitions of the same code smells and measurement strategies used by different tools, different threshold values embedded in these tools lead to the differences in resulting code smells [3].

Smell detection algorithms usually expose thresholds to engineers for customization [4]. Engineers may want to customize these thresholds for the following reasons. First, code smell identification is essentially subjective and application specific [5], [6]. For example, to identify *long methods* to *extract method*, detection algorithms should decide how long is *long*. Some software engineers may think 100 lines of source code must be the maximal length of a single method. Other engineers, however, may have better reading skills and thus prefer greater thresholds, e.g., 300. It is also possible that in applications whose business logic is simple and straightforward, the optimal threshold should be greater than usual. Thus, detection algorithms for *long methods* expose a threshold, e.g., *minLength*. Engineers may change this threshold manually according to their reading skills and the characteristics of the applications with which they are dealing.

Another reason why engineers should customize these thresholds is that they have different working schedules and different requirements on software quality, especially software maintainability. Code smell identification and software refactoring are time consuming, although they might result in software quality improvement. Consequently, software engineers from the industry must find out their own balance between refactoring costs and benefits according to their unique working schedules and quality requirements. For example, with plenty of time for quality improvement, they would prefer laxer criteria for smell detection that lead to lower precision and higher recall. In contrast, pushed by tight working schedules, they would prefer tighter criteria that lead to higher precision (and thus smaller refactoring costs) and lower recall. To achieve their unique balance between precision and recall, they must adjust thresholds of smell detection algorithms.

However, it is difficult for software engineers, especially inexperienced ones, to manually adjust thresholds in smell detection algorithms. For example, if the clone detection tool *KClone* [7] results in a low precision, e.g., less than 10 percent on a specific application, engineers may want to improve the precision to 50 percent with a minimal decrease in recall. They must manipulate the three parameters of *KClone*, i.e., *cg*, *dg*, and *w* [7]. However, the engineers with little experience in clone detection, might not know exactly which parameter(s) to change or how to change them. Engineers usually do not know the exact quantitative relations between thresholds' settings and performance, e.g., precision.

In summary, engineers should customize smell detection tools but a manual customization by adjusting thresholds in these tools is difficult. Thus, we propose an approach to

• *The authors are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China.*
E-mail: {Liuhui08, QiurongLiu, ZNiu, LiuYang}@bit.edu.cn.

Manuscript received 21 May 2014; revised 11 Oct. 2015; accepted 22 Nov. 2015. Date of publication 24 Nov. 2015; date of current version 20 June 2016.
Recommended for acceptance by A. Marcus.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2503740

adapt thresholds of smell detection tools automatically and dynamically. This paper offers the following contributions:

- First, we analyze the necessity of automatic threshold adaptation for code smell identification.
- Second, we propose an approach to adjust thresholds automatically and dynamically.
- Finally, we validate the proposed approach on open-source applications.

The rest of the paper is structured as follows. Section 2 presents a short overview of related research. Section 3 and Section 4 propose our feedback-based approach. Section 5 presents an evaluation of the proposed approach on open-source applications. Section 6 concludes.

2 Related Work

2.1 Code Smell Detection and Thresholds

The term *bad smells* was proposed by Fowler [8] in 1999. Bad smells, sometimes called *code smells*, are signs of potential problems in code that might require refactorings. Consequently, code smell identification is an important approach to identifying refactoring opportunities. Dozens of code smells have been well recognized by software engineers, e.g., *long method*, *duplicate code*, *feature envy*, and *refused bequest*. Dozens of algorithms and tools have also been proposed to identify these smells automatically or semi-automatically [1], [9].

Software metrics have been widely used to identify code smells. Marinescu [4] proposed detection strategies that are metrics-based rules to detect code smells. He defined and implemented such detection strategies for 14 important code smells. Lanza and Marinescu [10] defined metric-based detection rules for disharmonies (code smells), e.g., *God class*, *feature envy*, and *data class*. Munro [11] further developed the detection strategies by justifying the chosen metrics for code smells with additional empirical evidence. Similar work was conducted by Van Rompaey et al. [12] who proposed a metric-based approach for the detection of two test smells.

A number of heuristic algorithms have been proposed to identify code smells. Tsantalis and Chatzigeorgiou [13] proposed an algorithm to identify methods that are distributed in wrong locations (*feature envy*) and to solve the problems by using refactoring *move method*. Tsantalis and Chatzigeorgiou [14] proposed an algorithm to detect code smells that could benefit from the employment of polymorphism. Bavota et al. [15], [16] proposed an algorithm for the detection of classes that lack cohesion. They decomposed such classes by using *extract class* based on an analysis of relationships among methods. Tsantalis and Chatzigeorgiou [17] also proposed an algorithm for the detection of long methods and decomposed such methods by using *extract method*. A number of clone detection tools have been proposed including CCFinder [18], CBCD [19], Dup [20], Duploc [21], and KClone [7]. Evaluation and comparison among these tools have also been reported [22], [23]. Incremental clone detection algorithms are also available [24], [25].

Generic frameworks for code smell detection are also available. Tourwé and Mens [26] proposed a generic detection framework with which engineers can define smell detection algorithms with logic rules in SOUL, a logic programming language. Once compiled, the rules could be readily executed for smell detection. Moha et al. [27], [28] proposed a similar framework and a domain specific language to specify code smells.

All of these code smell detection algorithms discussed in preceding paragraphs use thresholds or may benefit from thresholds. For example, *CCFinder*, a widely used clone detection tool, has several thresholds exposed as parameters, e.g., *minimal Clone length* and *minimal TKS*. Metric-based detection algorithms [11], [12] use a threshold for each metric that they employ. KClone [7] exposes three thresholds: *cg*, *dg*, and *w*. Identification of classes that lack cohesion [15] has the threshold to remove all the edges (in the weighted graph representing a class) with a weight less than the fixed threshold. Fan-in based identification of crosscutting concerns [29] has a threshold to exclude methods with small fan-in. Although the feature envy detection approach proposed by Tsantalis and Chatzigeorgiou [13] does not have a threshold, it may be improved with a threshold to achieve balance between recall and precision. The approach computes for every *move method* suggestion the system-level Entity Placement (EP) metric that the system would have if the corresponding refactoring was applied. If a suggestion would lead to an EP value that is worse than the EP value of the current system, the approach would remove such suggestion. Otherwise, the suggestion would be presented to developers. However, minor improvement (or even no change) in EP may not compensate for the cost of refactorings. Consequently, setting a threshold in improvement of EP may help to improve the precision of the approach. Detection algorithms specified on generic detection frameworks [26], [27], [28] have thresholds as well. These algorithms are essentially the same as those manually designed for specific code smells. Consequently, thresholds required by traditional smell detection algorithms remain indispensable for those specified on generic frameworks.

Maiga et al. [30], [31] proposed a support vector machine (SVM) based approach to identify anti-patterns. From a given training data, their approach can learn how to identify similar anti-patterns. An advantage of their approach is that it does not depend on extensive knowledge of anti-patterns. They also use engineers' feedback to improve the accuracy by adopting an incremental SVM. Evaluation results suggest that engineers' feedback helps improve the accuracy of the anti-pattern detection. Our approach also takes advantage of engineers' feedback. Our work differs from theirs in that they learn from feedback to adapt the whole decision functions (SVM). In contrast, we learn from feedback to adapt the thresholds that are embedded in decision functions. While their work tries to replace existing approaches that are based on experts' knowledge of anti-patterns, our work improves such existing approaches with engineers' feedback.

Relative thresholds have been proposed to replace generic ones [32], [33]. Evaluation results reported by Crespo et al. [33] suggested that statistics of metrics, e.g., mean, bounded mean, standard deviation, lower quartile (Q1), median (Q2), upper quartile (Q3), minimum, and maximum vary from application to application. Consequently, to improve performance of code smell identification, thresholds should be application specific. Marinescu [32] suggested to use relative semantic thresholds like '*20 entities with the highest values*' and '10 percent of all measured entities having the lowest values', or to use statistical thresholds like BoxPlot, i.e., lower quartile (Q1), median (Q2), upper quartile (Q3). Relative thresholds, e.g., top 10 percent, are application specific. However, they do not take advantage of feedback from engineers. As a result, the performance, e.g., precision, varies from application to application.

2.2 Threshold Estimation for Smell Detection

Threshold setting has critical impact on the performance of code smell identification [4] and a number of approaches have been proposed to estimate/infer the optimal thresholds for code smell detection.

The first approach is to follow hints from literature. It is quite often that values of thresholds come from metrics' authors or the default values of widely used tools.

The second approach is to infer the optimal threshold values from a set of reference examples [6], [34]. Mihancea and Marinescu [34] called such approach *tuning machine*. Such *optimal* threshold setting that is inferred from reference applications is applied on other applications, hoping that it could lead to the same performance regardless of the differences in the applications to which it is applied and regardless of the differences in engineers who carry out the identification.

However, neither of the approaches discussed in the preceding paragraphs takes advantage of feedback from engineers for threshold adaptation. The two approaches generate generic thresholds for code smell identification algorithms. Such thresholds might not be optimal for specific applications or specific engineers.

Genetic algorithms (GA) have been successfully applied to search for optimal settings for algorithms in software engineering. For example, Panichella et al. [35] applied GA to determine a near-optimal configuration for Latent Dirichlet Allocation (LDA) [36]. Wang et al. [37] applied GA to search for threshold settings for clone detection tools to make a fair comparison among such tools. For given subject applications, they searched for settings that can maximize tool agreement on given subject applications. The work by Panichella et al. [35] and Wang et al. [37] suggested that GA could be effective in searching for optimal settings for algorithms in code smell identification. We also use a GA to search for the optimal threshold settings. Our work differs from the work by Panichella et al. [35] in that our work searches for threshold settings for code smell identification whereas Panichella et al. search for settings for LDA. Our work differs from the work by Wang et al. [37] in five aspects:

1) The two approaches solve different problems. Wang et al. tuned thresholds to improve the confounding configuration choice problem in comparison of different clone detection tools whereas we tune thresholds to facilitate engineers' customization of code smell identification. As a result, the fitness functions adopted in the two approaches are different, and the two approaches should be used in different scenarios.
2) Our approach takes advantages of feedback from developers that has not been considered by their approach.
3) Our approach works for all kinds of code smells whose identification algorithms require thresholds whereas Wang et al. focus on clone identification only.
4) Our approach works for a code smell detection algorithm (tool) regardless of the availability of alternative algorithms (tools). In contrast, their approach would not work if alternative tools are not available.
5) Our approach takes care of performance (precision and recall) of smell detection tools whereas their approach focuses on agreement among different clone detection tools. Consequently, tuning code smell detection tools with their approach may result in threshold settings where different smell detection tools agree well but their performance is poor, e.g., low precision. In contrast, our approach can avoid such cases because it maximizes recall whereas the actual precision is close to given target precision.

As discussed in Section 1, engineers should adapt thresholds for code smell identification tools according to different factors, e.g., characteristics of applications, expertise of engineers, working schedules, and quality requirements. It is difficult for engineers to adjust thresholds manually. Yet, to the best of our knowledge, there is no automatic approaches to adapting such thresholds dynamically with feedback from engineers. Thus, we propose a feedback-based approach to adapting thresholds automatically and dynamically. The ideal to leverage users' feedback to improve code smell identification has been mentioned briefly in our previous work [38]. However, the focus of the previous work [38] is a monitor-based instant refactoring framework, and thus it does not present the technical solution on how to adapt thresholds. It does not present the evaluation of the effect of feedback in improving code smell identification, either. In contrast, in this paper we present the technical solutions, i.e., how to formalize the optimization problem, how to estimate precision for given threshold settings, and how to solve the optimization problem with genetic algorithms. In this paper, we also present the evaluation of feedback-based threshold adaptation. The evaluation was carried out on five open-source applications involving four kinds of code smells, and results suggest that feedback-based threshold adaptation is effective in code smell identification.

3 Approach

3.1 Overview

An overview of the proposed approach is presented in Fig. 1. The approach is composed of two parts: Traditional code smell identification and refactoring approach, and threshold adaption approach. The traditional smell identification and refactoring part is composed of three stages that are linked in a loop: editing source code (software development), detecting code smells automatically, and checking code smells manually and applying refactorings (if necessary) to resolve the smells. The threshold adaption part is composed of two stages: collecting feedback and adapting threshold setting automatically. Both stages of the threshold adaption part are automatic and thus they could run in the background.

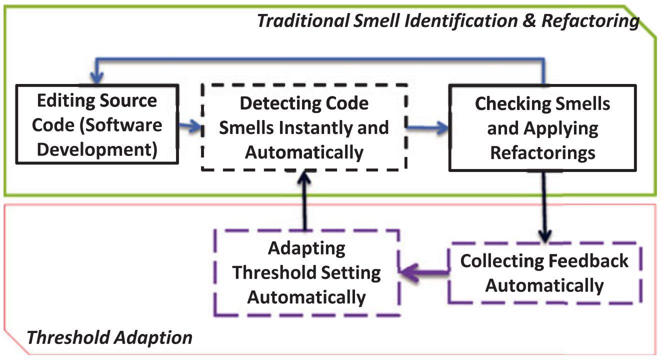

Traditional Smell Identification & Refactoring
Editing Source Code (Software Development)
Detecting Code Smells Instantly and Automatically
Checking Smells and Applying Refactorings
Adapting Threshold Setting Automatically
Collecting Feedback Automatically
Threshold Adaption

Fig. 1. Overview of the approach.

3.2 Detecting Smells Automatically

Smell detection algorithms employed in this stage could be any of the existing code smell detection algorithms that contain one or more thresholds. Thresholds of such smell detection algorithms could be initialized with default values provided by these algorithms. Approaches to infer default threshold settings have been introduced and discussed in Section 2.2. However, default settings that are inferred from reference samples rarely lead to best performance because code smell identification is subjective and application specific. Compared to traditional smell identification approaches where default threshold values are used all the time, the proposed threshold adaption depends less on the initial threshold values because it adjusts the thresholds dynamically and automatically.

3.3 Checking Smells and Applying Refactorings

In this stage, engineers manually check potential code smells recommended by smell detection algorithms and make decisions on which code smells should be resolved and which smells should be ignored. For those smells that must be resolved, engineers decide how to resolve them by refactorings and carry out these refactorings manually or semiautomatically with refactoring tools.

Manual checking of potential smells is indispensable even in traditional code smell identification and resolution. Smell identification is subjective and no approach can guarantee that no false positives would be recommended. Consequently, engineers must check the potential code smells manually before applying refactorings.

Manual checking of code smells is also indispensable for engineers to decide refactoring solutions. For example, given a long method, engineers should manually check it (with or without tool support) before they can decide which parts of the method should be extracted and where these parts should be moved.

3.4 Collecting Feedback Automatically

This stage collects feedback automatically from the engineers who manually check and resolve code smells. The proposed approach collects information about which code smells have been confirmed manually and which smells have been denied.

Recommended potential code smells are presented to engineers as a list and the proposed approach monitors changes to this list. For each smell in the list, engineers might (1) deny the potential code smell by removing this smell from the list; (2) resolve the code smell and it disappears automatically from the list; (3) mark the smell as 'to be resolved'. The first action suggests that the potential smell is manually denied whereas the last two actions suggest that this smell has been confirmed.

Thus, the collected feedback is composed essentially of a list of confirmed smells and a list of denied smells.

3.5 Adapting Threshold Setting Automatically

Threshold adaption is to improve performance of smell identification by changing threshold values according to feedback collected so far. Consequently, the adaption system is essentially a feedback controller. Details of the controller are presented in the following section.

4 Threshold Adaptation

In this section, we present the algorithm to adapt threshold setting automatically.

4.1 Goal of Threshold Adaptation

4.1.1 Available Metrics

Without specific working context (e.g., working schedule and specific quality requirements), threshold adaption should maximize metrics of the performance of smell identification, e.g., precision, recall, or F-Measure. Because precision and recall usually change in opposite directions, a synthesized measure, e.g., F-Measure and Matthews correlation coefficient (MCC), that combines both of them is an appropriate variable to maximize.

However, recall and synthesized measures that depend on it are not available for threshold adaptation because calculation of recall depends on the number of false negatives, i.e., true code smells that have been missed by smell detection tools. This number is not available in practice. Engineers check recommended potential smells (predicted positives) only and distinguish true positives from false positives. Consequently, what we can use to adapt thresholds is: the number of true positives (tp), the number of false positives (fp), and precision ($p = tp/(tp + fp)$). Smell identification tools cannot ask engineers to distinguish true negatives from false negatives and thus the number of true negatives is not available.

Yet, impact of thresholds' change on recall could be predicted although the concrete value of recall is not available. Wu et. al. [39] used delta recall to compare the recall of different approaches in identifying API changes while concrete values of recall are not available. We follow Wu et. al. and adopt delta recall to measure the impact of thresholds' change on recall. If changing threshold setting s_1 to s_2 makes the number of true positives change from tp_1 to tp_2, the recall would change from $r_1 = tp_1/N$ to $r_2 = tp_2/N$ where N is the number of actual code smells in the involved source code. Because N is unknown, the concrete value of recall, i.e., $r_1 = tp_1/N$ and $r_2 = tp_2/N$, is unknown. However, by comparing r_2 against r_1 we know how the recall has changed:

$$\begin{aligned}\Delta r &= \frac{r_2}{r_1}\\ &= \frac{tp_2/N - tp_1/N}{tp_1/N}\\ &= \frac{tp_2 - tp_1}{tp_1}\\ &= \frac{tp_2}{tp_1} - 1.\end{aligned} \tag{1}$$

4.1.2 Adaptation for Specific Working Schedule and Quality Requirements

As discussed in Section 1, engineers might want to adapt code smell identification algorithms to their unique working schedule and-or unique quality requirements. Thus, we expose a threshold (target precision, $targetP$) to engineers for manual customization. The threshold adaption maximizes recall while having precision close to the target precision. The goal of threshold adaptation is formalized as:

$$\begin{aligned}&\max r\\ &\text{subject to}\\ &\qquad p \geq targetP,\end{aligned} \tag{2}$$

where r and p are recall and precision, respectively.

We expose the threshold $targetP$ because of the following reasons:

1) First, with this threshold, engineers can adapt smell detection algorithms to their unique working schedules and quality requirements. On the one hand, they could increase the desired precision as a response to tight working schedules. On the other hand, they could decrease it as a response to higher quality requirements.
2) Second, it is easier for engineers to set a target precision than to change a set of contradicting and confusing thresholds because precision is a simple and common measure.
3) Finally, the proposed approach depresses engineers' intuitive desire for unpractically high precision by warning to what extent the recall would be reduced for the required improvement in precision. As discussed in Section 4.1.1 and Equation 1, we know to what extent recall would be reduced (e.g., by 80 percent) once the target precision is improved (e.g., increase from 60 to 90 percent). Consequently, whenever an engineer improves the target precision, the proposed approach warns with expected decrease in recall, e.g., *'recall would be reduced by 80 percent if the target precision is improved from 60 to 90 percent'*.

As a conclusion, engineers could adapt smell detection tools by controlling the target precision and this adaption is easier than that by manipulating confusing multi-thresholds directly.

4.2 Searching for the Optimal Threshold Setting

4.2.1 Simplified Optimization Problem

At time t, the proposed approach has collected a set of entities without smells reported by the smell identification algorithm ($Sclean$) and a set of entities with smells reported by the algorithm. The granularity of entities depends on the types of code smells. For example, while identifying *long methods*, we take individual methods as entities. In contrast, while identifying *large classes*, we take individual classes as entities. However, for a given type of code smells, only a single type of entities is collected. The set of entities with smells reported by the algorithm has been manually checked by engineers and it is decided into two sets: the set of entities whose smells have been accepted ($Saccepted$) and the set of entities whose smells have been rejected ($Srejected$). All entities (union of $Sclean$, $Saccepted$, and $Srejected$) are called $dataSet$. The actual recall and precision of smell identification (with threshold setting s) on $dataSet$ are called $r(s, dataSet)$ and $p(s, dataSet)$, respectively. The number of actual positives (code smells) in $dataSet$ is called $POS(dataSet)$. The number of true positives (with threshold setting s) in $dataSet$ is called $tp(s, dataSet)$.

The optimization problem specified by Equation 2 is equal to:

$$\max_{s} \; r(s, dataSet) \text{ subject to } p(s, dataSet) \geq targetP. \tag{3}$$

Because $r(s, dataSet) = tp(s, dataSet)/POS(dataSet)$ and $POS(dataSet)$ is independent of threshold setting s, Equation 3 is equal to:

$$\begin{aligned}&\max_s tp(s, dataSet)\\ &\qquad\text{subject to}\\ &\qquad\qquad p(s, dataSet) \geq targetP.\end{aligned} \tag{4}$$

4.2.2 GA-Based Searching Algorithm

There are a number of search algorithms for local or global optimization, such as hill climbing, heuristics, and genetic algorithms. We use GA to search for the optimal threshold setting because GA has the following advantages [40]: global optimization; robust; do not require derivative information or other auxiliary knowledge; mature implementation as both open-source applications and commercial products.

For the optimization problems specified by Equation 4, the fitness function is defined as follows:

$$f(s,dataSet) = \begin{cases} tp(s, dataSet) & \text{if } p(s, dataSet) \geq targetP\\ p(s, dataSet) & \text{others.}\end{cases} \tag{5}$$

Other parameters for GA are presented and discussed in Section 5.2.

4.2.3 Precision and True Positives

As discussed in Section 4.2.1, at time t, the proposed approach has collected a set of entities without smells reported by the smell identification algorithm ($Sclean$) and a set of entities with smells reported by the algorithm ($Ssuspicious$). $Ssuspicious$ has been manually checked by engineers and we know which entities are true positives ($Saccepted$) and which entities are false positives ($Srejected$). However, $Sclean$ has not been presented to

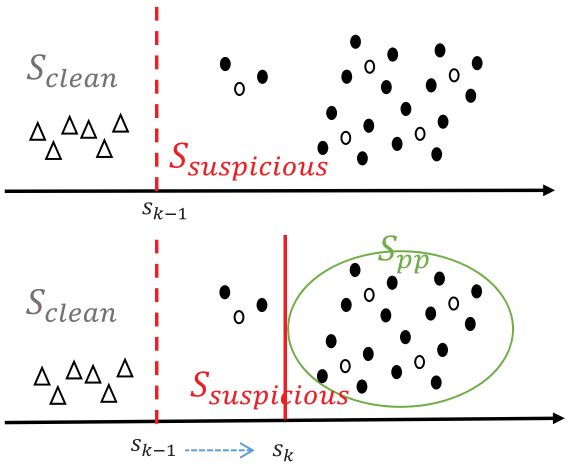


Fig. 2. Illustrating example ($Spp \subseteq Ssuspicious$).

engineers and thus we do not know which of them are true negatives or which of them are false negatives.

Changing threshold setting to s_k and reapplying the smell identification on the collected data $dataSet = Sclean \cup Ssuspicious$, we get a set of entities that are predicted as positives (predicated positives, notated as Spp). If $Spp \subseteq Ssuspicious$, i.e., all the recommended entities come from $Ssuspicious$ that has been manually checked, the number of true positives and precision can be calculated as follows:

$$tp(s_k, dataSet) = |Spp \cap Saccepted|, \tag{6}$$

$$p(s_k, dataSet) = \frac{tp(s_k, dataSet)}{|Spp|}. \tag{7}$$

An illustrating example is presented in Fig. 2. Hollow circles on the figure represent false positives, filled circles represent true positives, and triangles represent negatives (true negatives or false negatives). On the upper portion of the figure, the current threshold setting s_{k-1} separate $dataSet$ into $Sclean$ (6 triangles on the left side of the dotted segmentation line) and $Ssuspicious$ (23 circles on the right side). $Saccepted$ is composed of the 18 filled circles on the right side. When the threshold setting is changed from s_{k-1} to s_k, circles on the right hand side of the new segmentation line (the solid vertical line on the bottom portion of the figure) are predicted as positives (Spp). Since Spp is a subset of $Ssuspicious$ in this case, the precision could be calculated as Formula 7:

$$\begin{aligned} p(s_k, dataSet) &= \frac{tp(s_k, dataSet)}{|Spp|} \\ &= \frac{|Spp \cap Saccepted|}{|Spp|} \\ &= \frac{|16|}{20} \\ &= 0.8. \end{aligned} \tag{8}$$

However, $Spp \subseteq Ssuspicious$ might not always hold. It is possible that some entities in $Sclean$ are reported as suspicious (predicted positives) when threshold setting is changed to s_k, i.e., $Spp \cap Sclean \neq \emptyset$. We note such entities ($Spp \cap Sclean$) as $Sunchecked$. Because entities in $Sclean$ have not been manually checked, we do not know which

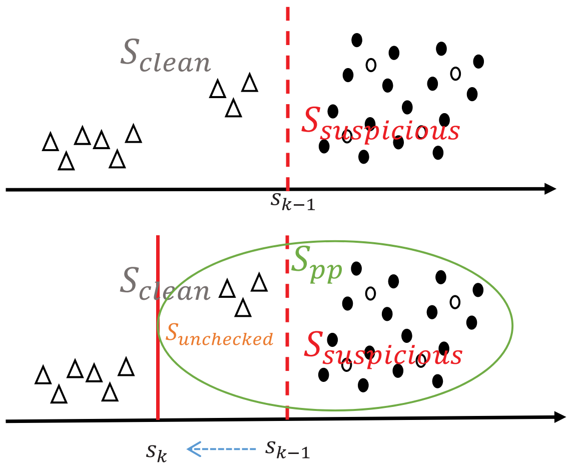


Fig. 3. Illustrating example where $Sunchecked = Spp \cap Sclean$ is not empty.

entities in $Sunchecked$ are true positives or which entities are false positives. As a result, we cannot calculate the number of true positives or precision accurately. In this case, we estimate the precision on $Sunchecked$ in the following way:

$$p(s_k, Sunchecked) = p(s_k, Schecked) \times e^{-\frac{|Sunchecked|}{|Schecked|}}, \tag{9}$$

where $Schecked = Spp \cap Ssuspicious$. When the size of $Sunchecked$ is small (approaching zero), the precision on it is close to that on $Schecked$. In contrast, when the size of $Sunchecked$ is large (approaching infinity), the precision on it is close to zero. The number of true positives in $Sunchecked$ could be calculated as:

$$tp(s_k, Sunchecked) = p(s_k, Sunchecked) \times |Sunchecked|. \tag{10}$$

As a result the number of true positives on $dataSet$ could be calculated as:

$$tp(s_k, dataSet) = tp(s_k, Sunchecked) + |Spp \cap Saccepted| \tag{11}$$

and precision is calculated as:

$$p(s_k, dataSet) = \frac{tp(s_k, dataSet)}{|Spp|}. \tag{12}$$

A corresponding example is presented in Fig. 3. When the threshold setting is changed from s_{k-1} to s_k, besides the whole set of $Ssuspicious$, three triangles ($Sunchecked$) are predicted as positives (Spp) as well. Consequently, Spp is not a subset of $Ssuspicious$. We first estimate the precision on $Sunchecked$ according to Formula. 9:

$$\begin{aligned} p(s_k, Sunchecked) &= p(s_k, Schecked) \times e^{-\frac{|Sunchecked|}{|Schecked|}} \\ &= \frac{16}{20} \times e^{-\frac{3}{20}} \\ &= 0.6886. \end{aligned} \tag{13}$$

The precision on the whole data set is then computed according to Formula 12:

$$\begin{aligned} p(s_k, dataSet) &= \frac{tp(s_k, dataSet)}{|Spp|} \\ &= \frac{tp(s_k, Sunchecked) + |Spp \cap Saccepted|}{23} \\ &= \frac{p(s_k, Sunchecked) \times |Sunchecked| + 16}{23} \\ &= \frac{0.6886 \times 3 + 16}{23} \\ &= 0.7855. \end{aligned} \tag{14}$$

5 Evaluation

In this section, we evaluate the proposed approach on open-source applications.

5.1 Research Questions

The evaluation investigates the following research questions:

- *RQ1*: How effective is the proposed approach compared against the *tuning machine* that infers optimal threshold values from a set of reference examples?
- *RQ2*: To what extend is the effect of the proposed approach influenced by initial values of thresholds?

Tuning machine has been introduced in Section 2.2. We compare the proposed approach against tuning machine because it is the only one that could respond automatically to customize target precision as the proposed approach does.

5.2 Configuration of GA

For the evaluation, we implement the proposed approach and tuning machine as a set of scripts based on the well-known GA toolbox *GATB*,[1] and the scripts are available on line at http://sei.pku.edu.cn/~liuhui04/data/scripts.7z.

The implementation of the proposed approach (and tuning machine as well) varies slightly according to different code smell detection algorithms because the number of thresholds required by such algorithms may vary. As an example, we present the implementation of the proposed approach for clone detection in Listing 1.

Most of the methods called in Listing 1, e.g, *crtbp*, *BS2RV*, and *ranking*, are provided by *GATB*. The only function that should be customized is *ObjectValue*, which computes the raw fitness for each individual according to the fitness function defined in Section 4.2.2.

The first input of the algorithm in Listing 1, i.e., $dataSet$, refers to the available data set collected so far. It is composed of S_{clean} and $S_{suspicisou}$ that are defined in Section 4.2.3. The second input, $target$, refers to the target precision. First of all, the algorithm initializes parameters required by *GATB*, including $NIND$ (number of individuals), $MAXGEN$ (maximum number of generations), $NVAR$ (number of variables), $PRECI$ (precision of variables), and $GGAP$ (generation gap). After that, it specifies the scope of variables (from Line 9 to Line 12). At Line 14, it generates the first population and computes the fitness for them at Line 17. In the following loop (from Line 18 to Line 34), the algorithm evolves until $MAXGEN$ generations are generated. In each loop, the algorithm computes the fitness for the entire population (Line 21) and selects the best individuals for breeding (Line 23). Crossover and mutation are applied at Line 25 and Line 27, respectively. After that, offspring is evaluated and inserted into the population, and the next loop begins. After the evolution, the algorithm selects the best individual from the population (Lines 35-37), decomposes the individual as two thresholds (Lines 38-39), and return such thresholds.

Listing 1. Adjusting thresholds for clone detection

```
1  function [newMCL, newMTKS] = gaAdjustThresholds
   (dataSet, target)
2    %dataSet is the data set collected so far (Ssuspicious is a subset
     of it).
3    %target is the target precision.
4      NIND = 20; %Number of individuals
5      MAXGEN = 1,000; %Maximum no. of generations
6      NVAR = 2; %No. of variables
7      PRECI = 9; %Precision of variables
8      GGAP = 0.8; %Generation gap
9      %Build field descriptor
10     FieldD1 = [PRECI; min(dataSet(:,1))-1; max(dataSet(:,1))
       +1; 0; 0; 1; 1];
11     FieldD2 = [PRECI; min(dataSet(:,2))-1; max(dataSet(:,2))
       +1; 0; 0; 1; 1];
12     FieldD = [FieldD1, FieldD2];
13     %Initialise population
14     Chrom = crtbp(NIND, NVAR * PRECI);
15     gen = 0; %Counter
16     %Evaluate initial population, call objective function defined
       in Section 4.2.2
17     ObjV = ObjectValue(dataSet, target, bs2rv(Chrom,
       FieldD));
18     %Generational loop
19     while gen < MAXGEN
20         %Assign fitness values to entire population
21         FitnV = ranking(ObjV);
22         %Select individuals for breeding
23         SelCh = select('sus', Chrom, FitnV, GGAP);
24         %Recombine individuals (crossover)
25         SelCh = recombin('xovsp', SelCh, 0.7);
26         %Apply mutation
27         SelCh = mut(SelCh, 0.03);
28         %Evaluate offspring, call objective function
29         ObjVSel = ObjectValue(dataSet, target, bs2rv(SelCh,
           FieldD));
30         %Reinsert offspring into population
31         [Chrom, ObjV] = reins(Chrom, SelCh, 1, 1, ObjV,
           ObjVSel);
32         %Increment counter
33         gen = gen + 1;
34     end
35     Chrom = bs2rv(Chrom, FieldD);
36     ObjV = ObjectValue(dataSet, target, Chrom);
37     [minValue, minIndex] = min(ObjV);
38     newMCL = Chrom(minIndex, 1);
39     newMTKS = Chrom(minIndex, 2);
40 end
```

Configuration of parameters used by *GATB* is presented in Table 1. As suggested by the table, the configurations for different smell detection algorithms are similar. The main difference is the *number of variables* that is equal to the

1. http://codem.group.shef.ac.uk/index.php/ga-toolbox

TABLE 1
Configuration of GA

Parameters	Long Method	Clone	Feature Envy	Inappropriate Intimacy
Number of individuals	20	20	20	20
Maximum no. of generations	1,000	1,000	1,000	1,000
No. of variables	3	2	1	1
Precision of variables	9	9	9	9
Generation gap	0.8	0.8	0.8	0.8
Selection function	sus	sus	sus	sus
Crossover function	xovsp	xovsp	xovsp	xovsp

number of thresholds used by specific detection algorithms. We use the stochastic universal sampling (sus) and single-point crossover (xovsp) because they are simple and effective [41].

5.3 Selected Code Smell Identification Algorithms

The evaluation is conducted on four code smell identification algorithms: detection algorithms for long methods, clones, feature envy, and inappropriate intimacy. All of such code smells are common in practice.

5.3.1 Long Method Identification

Long methods are those methods that are too long and too complex to read or maintain. A number of metrics have been used to detect long methods. The most frequently used metrics include Method Lines Of Code (MLOC), *Method Cyclomatic Complexity* (MCC), and slice based cohesion metrics [42]. We use slice based cohesion metrics (*tightness*, *coverage*, and *overlap*) to identify long methods that lack cohesion among statements within the same method. These metrics have been introduced by Weiser [43], formalized by Ott and Thuss [44], and validated by Meyers et al. [42]. The definition of these metrics are presented as:

$$Tightness(m) = \frac{|SL_{all}|}{length(m)} \tag{15}$$

$$Coverage(m) = \frac{1}{N} \times \Sigma_{i=1}^{N} \frac{|SL_i|}{length(m)} \tag{16}$$

$$Overlap(m) = \frac{1}{N} \times \Sigma_{i=1}^{N} \frac{|SL_all|}{SL_i}, \tag{17}$$

where m is a method, N is the number of slices in the method, SL_i is the ith slice, and SL_{all} is the number of statements in the intersection of all slices.

In the evaluation, these metrics are collected automatically by *JDeodorant* [45]. *JDeodorant* is selected because of the following reasons. First, it is publicly and freely available. Second, its source code is available (with academic license).

5.3.2 Clone Identification

There are a number of clone identification algorithms and tools [46]. We select *CCFinderX* [18] for evaluation for the following reasons. First, it is publicly available and it is free. Second, it is widely used and more than 5,000 licenses have been published (in request) [47].

CCFinderX contains two thresholds, *Minimum Clone Length* (MCL) and *Minimum TKS* (MTKS). Clone length is the length (in token) of a code fragment of the code clone. TKS (token set size) is the size of a set of tokens of a code fragment of the code clone. The default values of *MCL* and *MTKS* are 50 and 12, respectively.

5.3.3 Feature Envy

If one method is more interested in a class other than the one where it is defined, we say that there is a feature envy between the method and the class in which the method is interested [8]. Tsantalis [48] propose a distance-based approach to identify feature envy. For each method, it collects a set of the entities (S_m) that it accesses. For each class, it collects a set of entity (S_C) as well, i.e., all attributes and methods that belong to this class. Based on the sets, Tsantalis [48] defines the distance between a method m and a class C:

$$distance(m, C) = 1 - \frac{|S_m \cap S'_C|}{|S_m \cup S'_C|}, \tag{18}$$

where S'_C includes all methods and attributes belonging to this class except for m (if m belongs to class C).

If the distance between a method m and a class C_{envy} is significantly smaller than that between the method and its declaring class C_{def}, i.e., $distance(m, C_{def}) - distance(m, C_{envy}) > \beta$, a feature envy is reported.

5.3.4 Inappropriate Intimacy

One of the most famous design principles is low in coupling and high in cohesion. If two classes are highly coupled, we call it overintimate classes or inappropriate intimacy [8]. There are a number of metrics to measure the coupling between two classes [49], [50]. One of them is message passing coupling proposed by Li and Henry [51], which counts the number of messages sent among classes [52]. Based on this definition, we identify inappropriate intimacy by measuring the messages passing between each pair of classes:

$$\begin{aligned} MsgCoupling(c_1, c_2) = Number\,of\,messages\,passing \\ between\,c_1\,and\,c_2. \end{aligned} \tag{19}$$

If $MsgCoupling(c_1, c_2)$ is greater than a threshold ($MinMsgCoupling$), the pair of classes, i.e., c_1 and c_2, is reported as a potential inappropriate intimacy.

5.4 Data Collection and Oracles

To collect data (code smells) for evaluation, we ask three engineers to collect and analyze code smells from five open-source applications from SourceForge [53]. These applications are introduced as follows.

- *AutoFlight* is an AR.Drone control program with full sized video stream, navigation data display, and game controller input support. AR.Drone is a radio controlled flying helicopter built by the French company Parrot [54]. The application (version 0.056) contains 6,846 lines of source code and 332 methods. The source code is available at http://sourceforge.net/projects/autoflight/.
- *DirBuster* is a Java application designed to download directories and files on web and application servers. It aims to find out all pages and applications hidden within servers. The application (version 1.0) contains 12,929 lines of source code and 671 methods. The source code is available at http://sourceforge.net/projects/dirbuster.
- *Java3D Modeler* is a Java3D-based modeling tool. It allows users to create 3D models with primitives and texture sheets. The application (version 1.3.5) contains 9,105 lines of source code and 831 methods. The source code is available at http://sourceforge.net/projects/java3dmodeler.
- *PDF Split and Merge* is an open-source tool with graphical and command line interfaces. With this tool, users can spit, merge, mix, and rotate PDF documents easily. This applications (version 2.2.4) contains 11,383 lines of source code. The source code is available at https://sourceforge.net/projects/pdfsam/.
- *DavMail* is an email gateway. With this gateway, users may use any mail client to interact with Exchange. This application (version 4.5.1) contains 22,357 lines of source code. The source code is available at https://sourceforge.net/projects/davmail/.

To collect long methods, three engineers manually analyze every method in selected applications. However, it is challenging to manually check clones in non-trivial applications without clone detection tools because clone identification depends on comparison of different files. Consequently, the engineers collect a set of potential clones with *CCFinderX* that is configured conservatively (*MCL*=40 and *MTKS*=12) to achieve a high recall. With this setting, the number of predicted positives on selected applications increases by 95.4 percent compared to that when default setting is used. However, even with such a conservative setting, *CCFinderX* might miss some true clones, which is a threat to validity. A detailed discussion on this threat is presented in Section 5.7. They collect feature envy and inappropriate intimacy in the same way as they collect clones: Set thresholds conservatively ($\beta = 0$ and $minMsgCoupling = 3$) and manually check the resulting potential code smells. These data are available on line at http://sei.pku.edu.cn/~liuhui04/data/data.7z.

All of the involved engineers are graduate students majoring in computer science. They are familiar with code smells and software refactoring. The three engineers read applications together, and discuss before decisions are made. On most cases they can reach an agreement with a brief discussion. On other cases (accounting for no more than one percentage) they resort to a professor who is an expert on code smell identification and reach agreements finally with discussions.

5.5 Process

For each application (app) analyzed in Section 5.4, we carry out a k-fold ($k = 4$) cross-validation [55] by randomly partitioning the data set $dataSet$ collected from this application into four equally sized groups, notated as Sag_i ($i = 1 \dots 4$). The cross-validation process repeats four times. On each round of the cross-validation one of the four groups is used as training data ($trainingSet$), and the remaining three groups are used as the validation data ($testSet$). We train the approaches on one group instead of three groups (as traditional cross-validation testing dose) because such approaches are expected to learn from a small part of the subject application and to work on the larger part of it. If three-quarters of the application should be used to train the approaches which work for only one-quarter of the application, the value of such approaches would be small. For each code smell detection algorithm that are introduced in Section 5.3, we apply it to the $dataSet$ with default threshold setting and notate the resulting precision as p_d. We then compare related approaches, i.e., the proposed approach with inferred setting (*PAIS*), the proposed approach with default setting (*PADS*), and tuning machine (TM) by evaluating how effective they are in improving the precision by 20 percent on the $testSet$ of this application (app). The evaluation follows the following process.

First, we apply the tuning machine (*TM*) on $trainingSet$ to infer the optimal threshold setting s_{op} that leads to the greatest recall on $trainingSet$ while the precision is no less than target precision. Second, we apply the tuning machine with the optimal setting s_{op} to $testSet$. Third, we apply the proposed approach with the optimal setting s_{op} to $testSet$. Finally, we apply the proposed approach with default setting s_{op} to $testSet$.

5.6 Results

Results are presented in Tables 2, 3, 4, and 5. From these tables, we make the following observations:

1) The proposed approach outperforms tuning machine (RQ1). The actual precision of tuning machine has a great distance from corresponding target precision. The average distance is 0.115, and the maximal distance is 0.4. In contrast, the proposed approach makes the actual precision in close proximity to corresponding target. The average distance is reduced to 0.023, and the maximal distance is reduced to 0.082. In other words, the proposed approach reduces the distance between actual precision and target precision by 80% = (0.115-0.023)/0.115 on average compared to tuning machine.
2) Initial threshold values have little influence on the effect of the proposed approach (RQ2). From Tables 2, 3, 4, and 5 and Fig. 4, we observe that the proposed approach with default settings makes the actual precision close to targets as well as the approach with inferred settings does. With default settings, the maximal distance is 0.09 and the average distance is 0.032 whereas with inferred settings the maximal distance is 0.082 and the average distance is 0.023. These distances are much smaller than those when traditional tuning machine is employed.

TABLE 2
Results on Clone Detection

Application	Target Precision	K-Fold Cross-Validation	Actual Precision			Actual Recall		
			Tuning Machine	Proposed Approach (Inferred Setting)	Proposed Approach (default Setting)	Tuning Machine	Proposed Approach (Inferred Setting)	Proposed Approach (default Setting)
AutoFlight	51.4%	1#	45.8%	46.9%	48.1%	62.8%	56.7%	55.3%
		2#	61.5%	52.5%	49.5%	48%	50.4%	53.2%
		3#	62.5%	51.1%	50.6%	54.4%	54.6%	57%
		4#	34.6%	54.9%	50.4%	68.3%	32.9%	46.3%
		Average distance to target precision	0.109	0.024	0.018			
DirBuster	44.8%	1#	62.3%	42.1%	43.4%	81.1%	94%	80.8%
		2#	62.9%	44.9%	46.9%	81.5%	86.3%	85.4%
		3#	29.1%	51.2%	54.3%	73.9%	73.5%	70.7%
		4#	58.9%	43.7%	50.2%	84.3%	86%	85.3%
		Average distance to target precision	0.164	0.026	0.046			
Java3D Modeler	24.9%	1#	14.9%	22.9%	22.6%	78.6%	26.1%	26.1%
		2#	41.2%	28.3%	23.4%	38.9%	48.9%	63.3%
		3#	15.8%	20.0%	23.6%	69.0%	44.5%	42.6%
		4#	23%	24.4%	26.3%	55.6%	49.2%	41.7%
		Average distance to target precision	0.094	0.027	0.017			
DavMail	39%	1#	24.8%	34.3%	34%	81.3%	68.8%	68.8%
		2#	27.8%	37.8%	36.9%	50.9%	41.5%	43.6%
		3#	48.2%	31.6%	32.7%	49.1%	65.4%	65.4%
		4#	63.4%	38.2%	42.2%	41.7%	50.7%	45%
		Average distance to target precision	0.147	0.035	0.041			
PDF Split and Merge	21%	1#	13.7%	21.4%	18.3%	100%	69.2%	84.6%
		2#	36%	20.9%	15.2%	47.4%	49.4%	52.6%
		3#	18.3%	20.7%	21.7%	81.3%	75%	75%
		4#	18.5%	21.6%	20.6%	66.7%	61.1%	61.1%
		Average distance to target precision	0.069	0.003	0.024			

The results in Tables 2, 3, 4, and 5 suggest that in most cases the proposed approach with inferred setting outperforms the proposed approach with default setting. This may suggest that inferring threshold settings from training data is often valuable. However, the results also suggest that the distance between the performance of the two approaches (the proposed approach with inferred setting, and the proposed approach with default setting) is minor. In some

TABLE 3
Results on Long Method Detection

Application	Target Precision	K-Fold Cross-Validation	Actual Precision			Actual Recall		
			Tuning Machine	Proposed Approach (Inferred Setting)	Proposed Approach (default Setting)	Tuning Machine	Proposed Approach (Inferred Setting)	Proposed Approach (default Setting)
AutoFlight	60%	1#	78.1%	59.4%	56.4%	56.8%	73%	75.9%
		2#	56%	57.6%	57.9%	68.3%	63.9%	66.6%
		3#	56%	57.8%	57.1%	70%	65%	65.3%
		4#	44.6%	53%	53.6%	80.7%	69%	67.1%
		Average distance to target precision	0.104	0.031	0.037			
DirBuster	68.6%	1#	61.2%	64.5%	63.6%	68.3%	67.3%	66.8%
		2#	83.3%	67.4%	64.8%	59.7%	66.7%	67%
		3#	59.1%	63%	62.8%	70%	69.2%	69.8%
		4#	54.6%	64.1%	62.9%	67.7%	64.7%	64.2%
		Average distance to target precision	0.114	0.038	0.050			
Java3D Modeler	51%	1#	47.7%	47.9%	48.7%	64.6%	72.1%	70.2%
		2#	56.5%	50.9%	50.3%	69.6%	73.4%	72.7%
		3#	39%	45.4%	45.3%	68.5%	55.9%	58.5%
		4#	38.2%	45.3%	44.9%	75%	53.7%	58.7%
		Average distance to target precision	0.084	0.036	0.037			
DavMail	46.5%	1#	29.2%	38.5%	37.5%	38.9%	27.8%	33.3%
		2#	42.9%	46.7%	47.1%	26.1%	21.7%	19.8%
		3#	50%	46.1%	45.8%	29.2%	41.7%	42.3%
		4#	35.5%	45.7%	44.8%	57.9%	42.1%	42.1%
		Average distance to target precision	0.088	0.023	0.030			
PDF Split and Merge	51.9%	1#	48%	52.4%	50%	48%	44%	48%
		2#	81.8%	54.6%	52.2%	30%	40%	40%
		3#	62.5%	52.2%	44%	35.7%	39.3%	42.9%
		4#	39.4%	50%	43.5%	46.4%	32.1%	35.7%
		Average distance to target precision	0.142	0.013	0.046			

TABLE 4
Results on Feature Envy Detection

Application	Target Precision	K-Fold Cross-Validation	Actual Precision			Actual Recall		
			Tuning Machine	Proposed Approach (Inferred Setting)	Proposed Approach (default Setting)	Tuning Machine	Proposed Approach (Inferred Setting)	Proposed Approach (default Setting)
AutoFlight	27.5%	1#	40.6%	25.5%	24.6%	72.2%	77.8%	88.9%
		2#	21.2%	25.9%	30%	93.3%	93.3%	86.7%
		3#	20.8%	25.4%	29.3%	100%	100%	80%
		4#	38.1%	26.2%	30.4%	88.9%	94.4%	94.4%
		Average distance to target precision	0.092	0.018	0.024			
DirBuster	35.6%	1#	43.4%	33.9%	35.4%	94.7%	97.4%	96.2%
		2#	64.3%	40.5%	44.6%	75%	88.9%	80.6%
		3#	43.4%	33.4%	32.5%	94.7%	100%	100%
		4#	55.6%	35.4%	39.1%	71.4%	80%	77.1%
		Average distance to target precision	0.160	0.023	0.040			
Java3D Modeler	35.4%	1#	41.1%	36.9%	33.0%	93.8%	96.9%	100%
		2#	25.2%	33.3%	33.8%	100%	92.3%	88.5%
		3#	53.7%	38.0%	37.0%	66.7%	90.9%	90.9%
		4#	39.5%	34.4%	33.7%	93.8%	93.8%	96.9%
		Average distance to target precision	0.095	0.018	0.018			
DavMail	61%	1#	75.4%	61.1%	60%	93.8%	96.3%	98%
		2#	57.9%	59.8%	57.9%	97.8%	96%	97.8%
		3#	58.6%	61.5%	60.8%	95.3%	85.4%	93%
		4#	51.9%	58.4%	56.3%	97.7%	86.8%	93%
		Average distance to target precision	0.073	0.011	0.023			
PDF Split and Merge	39.7%	1#	73.5%	38.8%	38%	83.3%	95.6%	97.8%
		2#	51.8%	40.2%	39.4%	96.7%	97.8%	98.9%
		3#	43.6%	40.1%	40.3%	96.4%	97.6%	96.5%
		4#	34.9%	43.3%	42.5%	100%	93.1%	95.4%
		Average distance to target precision	0.137	0.013	0.014			

cases, the proposed approach with default setting can even outperform the proposed approach with inferred setting. One of the reasons is that the initial threshold values have little influence on the effect of the proposed approach. The proposed approach dynamically changes the threshold setting according to its changing precision, and makes the actual precision close to the given target precision. While more and more history data are available, the actual

TABLE 5
Results on Inappropriate Intimacy Detection

Application	Target Precision	K-Fold Cross-Validation	Actual Precision			Actual Recall		
			Tuning Machine	Proposed Approach (Inferred Setting)	Proposed Approach (default Setting)	Tuning Machine	Proposed Approach (Inferred Setting)	Proposed Approach (default Setting)
AutoFlight	72.9%	1#	87.5%	76.9%	73.3%	43.8%	62.5%	68.8%
		2#	69.2%	72.7%	72.7%	60%	53.3%	53.3%
		3#	85.7%	70%	66.7%	46.2%	53.9%	61.5%
		4#	52.4%	71.4%	81.8%	84.6%	76.9%	69.2%
		Average distance to target precision	0.129	0.021	0.039			
DirBuster	41.5%	1#	64%	41.2%	36.7%	64%	84%	88%
		2#	61.9%	40.5%	42.4%	59.1%	68.2%	63.6%
		3#	36.2%	39.5%	42.1%	85%	85%	80%
		4#	31.7%	44.1%	34%	95%	75%	90%
		Average distance to target precision	0.145	0.015	0.035			
Java3D Modeler	73%	1#	54.6%	70.4%	77.3%	88.2%	55.9%	50%
		2#	75.8%	71.4%	71.4%	69.4%	83.3%	83.3%
		3#	63.9%	73.9%	70.4%	76.7%	56.7%	63.3%
		4#	88.9%	75%	80.7%	42.1%	79%	65.8%
		Average distance to target precision	0.116	0.018	0.04			
DavMail	61%	1#	75%	62.1%	61.1%	50%	60%	65%
		2#	52.4%	57.1%	57.9%	73.3%	66.7%	66.7%
		3#	54.6%	60.9%	60%	80%	46.7%	50%
		4#	73.7%	67.7%	62.5%	43.8%	65.6%	66.7%
		Average distance to target precision	0.104	0.029	0.014			
PDF Split and Merge	62.4%	1#	67.9%	60.6%	58.8%	63.3%	66.7%	66.9%
		2#	73.3%	61.7%	58%	66.7%	87.9%	89.9%
		3#	49.2%	54.3%	57.6%	96.7%	38.3%	33.3%
		4#	48.2%	62.9%	65.7%	100%	81.5%	79%
		Average distance to target precision	0.1	0.028	0.04			

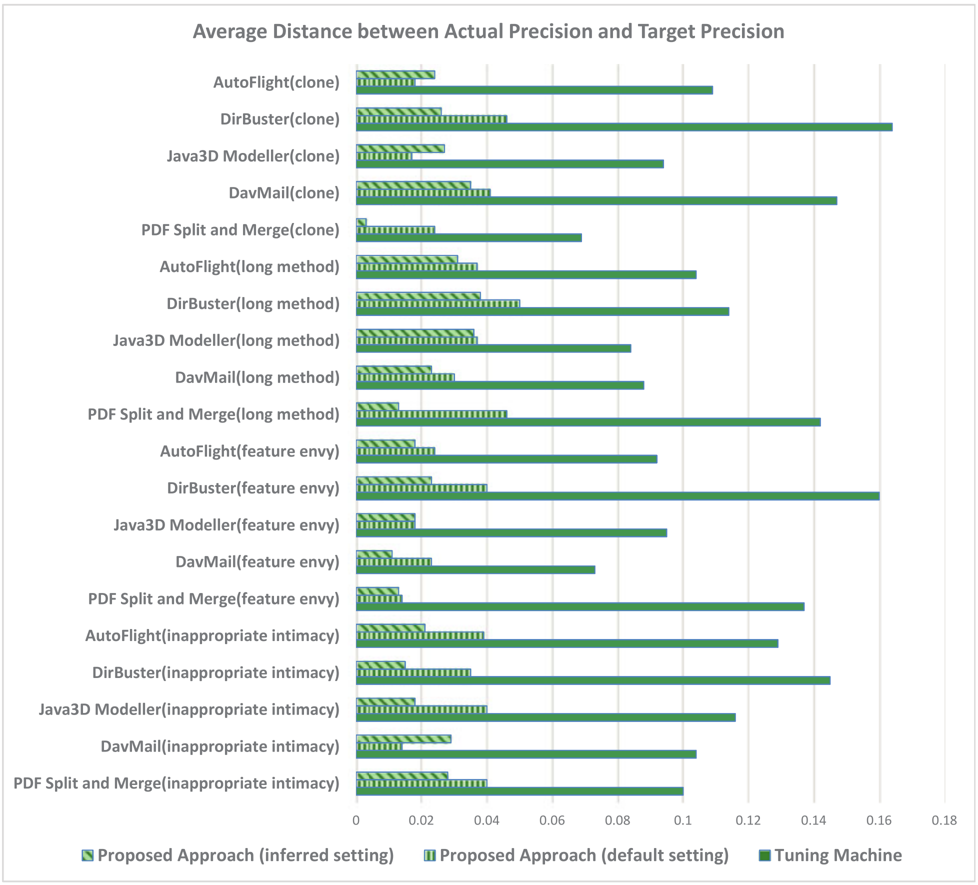


Fig. 4. Average distance from target precision.

threshold values become stable. As a result, the impact of the initial threshold values on the final precision is minimized. However, the initial threshold values have greater impact on decisions of the first few potential smells when the threshold values are changing frequently. Considering the great randomness in the distribution of the metrics of the first few potential smells, the default threshold values may happen to be better suited to such (a small number of) smells than the inferred threshold values. As a result, in some cases the default values may result in better performance than the inferred values, although the distance is usually very small and could be ignored.

The results in Fig. 4 also suggest that the effect (measured with distance between actual precision and target precision) of the involved approaches, both the proposed approaches and tuning machine, varies from application to application. One possible reason is the randomness in participating data sets randomly during the k-fold cross-validation introduced in Section 5.5. The more representative the training sets are, the easier for the involved approaches to approach the target precision. Another possible reason for the variation is that the distribution of the metrics of involved smells may vary from application to application. Maldistribution of such metrics where the metrics vary dramatically among different code smells may increase the difficulty for the approaches to control actual precision. In contrast, even distribution of such metrics may facilitate the control. The third possible reason for the variation is that the strength of the correlation between involved metrics (where thresholds are applied) and the decisions on code smells varies from application to application. When the correlation is stronger, it is easier to control the actual precision by changing thresholds, and thus the effect of the involved approaches is more obvious. However, when the correlation is weaker, it is more difficult to control the actual precision by changing the thresholds only. One of the possible solutions to the weak correlation is to use different metrics for code smell identification in different applications. However, selection of metrics for code smell identification is out of the scope of this paper. This limitation is also discussed in the second and third paragraphs of Section 5.8. Despite the variation in effect, however, the proposed approach outperforms tuning machine on all subject applications and on all kinds of code smells.

5.7 Threats to Validity

A threat to external validity is that the evaluation is conducted on four smell detection algorithms only. Reasons why these algorithms are selected have been presented in Section 5.3. To improve the validity, however, we should carry out further evaluation on other algorithms, e.g., detection algorithms for *large class*.

The second threat to external validity is that the evaluation is conducted on five applications only. Special characteristics of the applications may have biased the evaluation results. The reason why these applications are selected is discussed in Section 5.4. To make the conclusions more reliable, further evaluation on more applications should be conducted in future.

The third threat to external validity is that participants of the evaluation are graduate students. Conclusions drawn from students are not necessarily true for experienced software engineers. We select these graduate students because of the following reasons. First, these students are familiar with code smell identification and software refactoring. Second, all of them have more than three years' experience in software development in Java. Third, it is challenging to get experienced software engineers from the industry for the identification of all code smells in open-source applications. They prefer to work on their own applications but proprietary applications would make the experiment hard to replicate.

A threat to construct validity is that manual identification is subjective. As introduced in Section 1, code smell identification is necessary subjective. Consequently, different people may make different decisions on the same source code. As a result, some special characters of the participants may have yielded the results. To reduce the threat, we have taken the following measures. First, we assign three persons as a group, and they should discuss before final decisions are made, which may help to reduce the subjectiveness. Second, we have not told the participants the purpose of the evaluation, which may reduce potential subjective bias towards the proposed approach. Third, we make the implementation of the proposed approach publicly available, and thus other researchers may validate it with code smells identified by other people.

The second threat to construct validity is that manual identification might not identify all clones and thus the recall calculated in the evaluation might be inaccurate. As stated in Section 5.3.2, clones are collected with the help of *CCFinderX* that might not retrieve all clones. To reduce the threat, *CCFinderX* is configured conservatively to reduce the number of missed clones (i.e. to increase the number of true positives). Although such conservative setting succeeds in increasing the number of true positives by 52 percent compared to that when default setting is used, it doubles the number of false positives, and thus makes the manual checking more difficult.

The third treat to construct validity is that our implementation of tuning machine might be buggy. As introduced in Section 5.1, the proposed approach is compared against tuning machine. However, we fail to find its implementation, and thus we re-implement the algorithm. It is possible that our implementation is buggy and it does not perform as expected, which is a threat to the validity. To this end, we make the data set publicly available, and thus other researchers can re-implement the algorithms and repeat the evaluation.

Another threat to construct validity is that the code smells' detection algorithms involved in the evaluation might not be the best ones. For example, the identification of long methods based on cohesion metrics only might be less accurate than those based on *MLOC*, *MCC*, and cohesion metrics. However, the focus of the evaluation is to evaluate the effect of the proposed approach instead of the performance of different code smell detection algorithms. Consequently, the evaluation prefers simple, effective smell detection algorithms. However, in future the proposed approach should be evaluated on more complex smell detection algorithm.

5.8 Limitations

The first limitation of the proposed approach is that it cannot be generalized to all code smell detection or refactoring recommendation approaches. It works well for metric-based techniques that require thresholds to operate. However, the detection of some code smells may depend on pure static source analysis (or lexical analysis) and use no threshold. For example, the detection of *public fields* and *bad names* usually uses no thresholds, and thus the proposed approach does not work for such code smells.

Another limitation of the proposed approach is that it does not optimize smell detection algorithms except for their threshold values. The proposed approach considers such algorithms as black boxes that expose thresholds only. Consequently, even if adapting the exposed thresholds cannot make the actual precision close to the target precision, the proposed approach cannot adapt the algorithms themselves. For example, if the *long method* detection algorithm detects long methods according to methods' length (*LOC*) only and the target precision is high (e.g., 90 percent), it is likely that its actual precision is lower than the target precision even though the threshold has been optimized by the proposed approach. In this case, however, the proposed approach will not automatically change the algorithm itself (e.g., filtering out false positives with additional metrics) to improve actual precision. How often the approach may fail to make actual precision close to given target precision depends on the code smell detection algorithms, involved applications, and the target precision.

The limitations discussed in the preceding paragraphs come from the fact that feedback from engineers is used to adapt thresholds only. However, it could be possible to adapt other aspects of detection algorithms. As a result, the discussed limitations may disappear. For example, although the detection of *public fields* does not require thresholds, feedback from engineers, i.e., whether the engineers take public fields as *bad smells*, can be used to adapt the detection: we may stop warning engineers with public fields if they refuse to encapsulate such fields. Feedback from engineers may also be used to switch automatically among different algorithms that identify the same code smells.

6 Conclusions and Future Work

In this paper, we first highlight the necessity of dynamic and automatic threshold adaption for code smell detection. We then propose a feedback-based approach to adapt such thresholds, and evaluate the proposed approach on open-source applications. Evaluation results suggest that the proposed approach is effective and it outperforms the traditional tuning machine.

Further evaluation should be carried out on more code smell detection algorithms. More subject applications and experienced engineers from the industry should be involved in further evaluation as well. It is also interesting to apply the proposed approach to other Recommendation Systems for Software Engineering (RSSEs) [56] that should be customized by developers as well.

Acknowledgments

The authors would like to say thanks to the associate editor and the anonymous reviewers for their valuable and constructive suggestions. The work is funded by the National Natural Science Foundation of China (No. 61272169, 61472034), Program for New Century Excellent Talents in University (No. NCET-13-0041), and Beijing Higher Education Young Elite Teacher Project (No. YETP1183). Zhendong Niu is supported by the National Natural Science Foundation of China (No. 61370137), the National Strategic Basic Research Program ("973" Program) of the Ministry of Science and Technology of China (Grant No. 2012CB720702), and the 111 Project of Beijing Institute of Technology.

REFERENCES

[1] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Software Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.

[2] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, Univ. Illinois, Urbana-Champaign, IL, USA, 1992.

[3] D. Bowes, D. Randall, and T. Hall, "The inconsistent measurement of message chains," in *Proc. 4th Int. Workshop Emerging Trends Softw. Metrics*, May 2013, pp. 62–68.

[4] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. 20th IEEE Int. Conf. Softw. Maintenance*, 2004, pp. 350–359.

[5] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, "Are automatically-detected code anomalies relevant to architectural modularity? an exploratory analysis of evolving systems," in *Proc. 11th Annu. Int. Conf. Aspect-Oriented Softw. Develop.*, 2012, pp. 167–178.

[6] K. A. M. Ferreira, M. A. S. Bigonha, R. S. Bigonha, L. F. O. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *J. Syst. Softw.*, vol. 85, no. 2, pp. 244–257, Feb. 2012.

[7] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita, "KClone: A proposed approach to fast precise code clone detection," in *Proc. 3rd Int. Workshop Softw. Clones*, Mar. 2009, http://www.informatik.uni-bremen.de/st/IWSC/

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.

[9] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," *J. Softw. Maintenance Evolution: Res. Practice*, vol. 23, no. 3, pp. 179–202, 2011.

[10] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. New York, NY, USA: Springer, 2006.

[11] M. Munro, "Product metrics for automatic identification of "bad smell" design problems in Java source-code," in *Proc. 11th IEEE Int. Symp. Softw. Metrics*, Sep. 2005, p. 15.

[12] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 800–817, Dec. 2007.

[13] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Softw. Eng.*, vol. 35, no. 3, pp. 347–367, May/Jun. 2009.

[14] N. Tsantalis and A. Chatzigeorgiou, "Identification of refactoring opportunities introducing polymorphism," *J. Syst. Softw*, vol. 83, no. 3, pp. 391–404, 2010.

[15] G. Bavota, A. D. Lucia, and R. Oliveto. (2011). Identifying extract class refactoring opportunities using structural and semantic cohesion measures, *J. Syst. Softw.* [Online]. *84(3)*, pp. 397–414. Available: http://www.sciencedirect.com/science/article/pii/S0164121210003195

[16] G. Bavota, R. Oliveto, A. De Lucia, G. Antoniol, and Y. Gueheneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–5.

[17] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities for the decomposition of methods," *J. Syst. Softw.*, vol. 84, pp. 1757–1782, Oct. 2011.

[18] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multi-linguistic token based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 6, pp. 654–670, Jul. 2002.

[19] J. Li and M. D. Ernst, "CBCD: Cloned buggy code detector," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 310–320.

[20] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. 2nd IEEE Working Conf. Rev. Eng.*, Jul. 1995, pp. 86–95.

[21] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. Int. Conf. Softw. Maintenance*, 1999, pp. 109–118.

[22] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 577–591, Sep. 2007.

[23] C. K. Roy, J. R. Cordy, and R. Koschke. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Sci. Comput. Program.* [Online]. *74*, pp. 470–495. Available: http://www.sciencedirect.com/science/article/pii/S0167642309000367

[24] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: Incremental, distributed, scalable," in *Proc. IEEE Int. Conf. Softw. Maintenance*, Sep. 2010, pp. 1–9.

[25] N. Gode and R. Koschke, "Incremental clone detection," in *Proc. 13th Eur. Conf. Softw. Maintenance Reeng.*, 2009, pp. 219–228.

[26] T. Tourwé and T. Mens, "Identifying refactoring opportunities using logic meta programming," in *Proc. 7th Eur. Conf. Softw. Maintenance Reeng.*, 2003, pp. 91–100.

[27] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan./Feb. 2010.

[28] N. Moha, Y.-G. Gueheneuc, and P. Leduc, "Automatic generation of detection algorithms for design defects," in *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Sep. 2006, pp. 297–300.

[29] M. Marin, A. V. Deursen, and L. Moonen, "Identifying crosscutting concerns using fan-in analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 1, pp. 3:1–3:37, 2007.

[30] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. Gueheneuc, and E. Aimeur, "Smurf: A SVM-based incremental anti-pattern detection approach," in *Proc. 19th Working Conf. Reverse Eng.*, Oct. 2012, pp. 466–475.

[31] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. Gueheneuc, G. Antoniol, and E. Aimeur, "Support vector machines for anti-pattern detection," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Sep. 2012, pp. 278–281.

[32] R. Marinescu, "Detecting design flaws via metrics in object-oriented systems," in *Proc. 39th Int. Conf. Exhibition Technol. Object-Oriented Lang. Syst.*, 2001, pp. 173–182.

[33] Y. Crespo, C. López, and R. Marticorena, "Relative thresholds: Case study to incorporate metrics in the detection of bad smells," in *Proc. 10th ECOOP Workshop Quantitative Approaches Object-Oriented Softw. Eng.*, Jul. 2006, pp. 109–118.

[34] P. Mihancea and R. Marinescu, "Towards the optimization of automatic detection of design flaws in object-oriented software systems," in *Proc. 9th Eur. Conf. Softw. Maintenance Reeng.*, 2005, pp. 92–101.

[35] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. (2013). How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms, in *Proc. Int. Conf. Softw. Eng* [Online]. pp. 522–531. Available: http://dl.acm.org/citation.cfm?id=2486788.2486857

[36] D. M. Blei, A. Y. Ng, and M. I. Jordan. (2003, Mar.). Latent Dirichlet allocation, *J. Mach. Learn. Res.* [Online]. *3*, pp. 993–1022. Available: http://dl.acm.org/citation.cfm?id=944919.944937

[37] T. Wang, M. Harman, Y. Jia, and J. Krinke. (2013). Searching for better configurations: A rigorous approach to clone evaluation, in *Proc. 9th Joint Meeting Foundations Softw. Eng.* [Online]. pp. 455–465. Available: http://doi.acm.org/10.1145/2491411.2491420

[38] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1112–1126, Aug. 2013.

[39] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "AURA: A hybrid approach to identify framework evolution," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.- vol. 1*, 2010, pp. 325–334.

[40] R. Poli, W. B. Langdon, and N. F. McPhee, *A Field Guide to Genetic Programming*. Lulu Enterprises, U.K. Ltd, 2008.

[41] A. Chipperfield, P. Fleming, H. Pohlheim, and C. Fonseca, "Genetic algorithm toolbox user's guide," Dept. Autom. Control Syst. Eng., Univ. Sheffield, [Online] Available: http://codem.group.shef.ac.uk/public/GAToolbox_Documentation.pdf

[42] T. M. Meyers and D. Binkley, "An empirical study of slice-based cohesion and coupling metrics," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 1, pp. 2:1–2:27, Dec. 2007.

[43] M. Weiser. (1981). Program slicing, in *Proc. 5th Int. Conf. Softw. Eng.* [Online]. pp. 439–449. Available: http://dl.acm.org/citation.cfm?id=800078.802557

[44] L. Ott and J. Thuss, "Slice based metrics for estimating cohesion," in *Proc. 1st Int. Softw. Metrics Symp.*, May 1993, pp. 71–81.

[45] JDeodorant. (2014). [Online]. Available: http://java.uom.gr/~jdeodorant/

[46] R. Koschke. (2007). Survey of research on software clones [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/962

[47] (2014). [Online]. Available: http://www.ccfinder.net/index.html

[48] N. Tsantalis, "Evaluation and improvement of software architecture: Identification of design problems in object-oriented systems and resolution through refactorings," PhD, Dept Applied Informatics, Univ. Macedonia, Thessaloniki, Greece, Aug. 2010.

[49] N. E. Fenton, *Software Metrics: A Rigorous Approach*. London, U.K.: Chapman & Hall, 1991.

[50] L. Briand, J. Wust, and H. Lounis, "Using coupling measurement for impact analysis in object-oriented systems," in *Proc. IEEE Int. Conf. Softw.*, 1999, pp. 475–482.

[51] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *J. Syst. Softw.*, vol. 23, no. 2, pp. 111–122, Nov. 1993.

[52] R. Lincke and W. Löwe. (2007, Apr. 4). Compendium of software quality standards and metrics (version 1.0), Nikos Drakos, Computer Based Learning Unit, University of Leeds. Ross Moore, Mathematics Department, Macquarie University, Sydney. [Online]. Available: http://www.arisa.se/compendium/quality-metrics-compendium.html

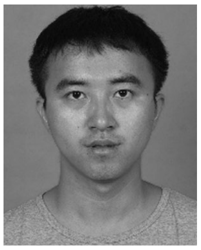
[53] SourceForge. (2014). [Online]. Available: http://sourceforge.net/

[54] Drone. (2014). [Online]. Available: http://ardrone2.parrot.com/

[55] S. Geisser, *Predictive Inference*. London, U.K.: Chapman & Hall, 1993.

[56] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Softw.*, vol. 27, no. 4, pp. 80–86, Jul./Aug. 2010.

Hui Liu received the BS degree in control science from the Shandong University in 2001, the MS degree in computer science from the Shanghai University in 2004, and the PhD degree in computer science from the Peking University in 2008. He is an associate professor at the School of Computer Science and Technology, Beijing Institute of Technology. He is particularly interested in software refactoring, design pattern, and software evolution. He is currently doing research to make software refactoring easier and safer. He is also interested in developing practical refactoring tools to assist software engineers. He is a member of the IEEE.



Qiurong Liu received the BS degree in information and computing science from the Beijing Jiaotong University in 2011, then worked two years as a software engineer. He is currently working toward the Master's degree at the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. He is interested in software refactoring and software evolution.

Zhendong Niu received the PhD degree in computer science from the Beijing Institute of Technology, Beijing, China, in 1995. He was a Post-Doctoral Researcher with the University of Pittsburgh, Pittsburgh, PA, from 1996 to 1998, a Research/Adjunct Faculty Member with the Carnegie Mellon University, Pittsburgh, from 1999 to 2004, and a Joint Research Professor with the Information School, University of Pittsburgh, in 2006. He is a professor and the deputy dean with the School of Computer Science and Technology, Beijing Institute of Technology. His current research interests include informational retrieval, software architecture, digital libraries, and Web-based learning techniques. He is a recipient of the IBM Faculty Innovation Award in 2005 and the New Century Excellent Talents, University of Ministry of Education, China, in 2006.

Yang Liu received the BS degree in computer science from the Beijing University of Chemical Technology in 2012, and the MS degree in software engineering from the Beijing Institute of Technology in 2015 under the supervision of Dr. Hui Liu. She is interested in software refactoring, software testing, and software evolution.