AdaBoost-based Refused Bequest Code Smell Detection with Synthetic Instances

Hao Chen*[1], Zhilei Ren*[3], Lei Qiao†[2], Zhide Zhou*[5], Guojun Gao*[7], Yue Ma‡[6], He Jiang*[4]
*School of Software, Dalian University of Technology, Dalian, China
†Beijing Institute of Control Engineering, Beijing, China
‡Taiyuan University of Technology, Taiyuan, China
{[1]chenhaodlut, [2]fly2moon}@163.com, {[3]zren, [4]jianghe}@dlut.edu.cn
{[5]cszide, [6]csyuema}@gmail.com, [7]ggj_gao@mail.dlut.edu.cn

***Abstract*—Software requirements are constantly changing. Consequently, the development process is frequently under time pressure, which results in technical debt. To illustrate the symptoms of technical debt, 22 code smells have been introduced to indicate the poor design in code fragment, among which refused bequest is one of the most harmful smells and with high diffuseness. However, refused bequest is rarely taken into account because there is a lack of dataset. Moreover, it is difficult to design the detection rules for refused bequest compared with other popular smells.**

In this paper, we propose a machine-learning-based refused bequest smell detection framework SEADART, which features the utilization of a set of synthetic smelly instances. Specifically, SEADART comprises three components: (1) a smell generation approach, and (2) a model training strategy, and (3) an AdaBoost-based detection model. We evaluate the performance of the proposed framework. The evaluation results suggest that the generated smelly instances are reliable, and the trained AdaBoost model significantly outperforms the state-of-the-art over a real-world dataset.

***Index Terms*—Technical Debt, Code Smell, Refused Bequest, AdaBoost**

I. Introduction

During the software life cycle, it takes developers a large amount of time to conduct software maintenance and evolution, in order to meet the constantly changing requirements from users [1]. These processes are frequently performed under time pressure, resulting in poor programming design or implementation applied by developers, which is so-called technical debt [2]. To illustrate the symptoms of technical debt, Beck and Fowler [3] introduce the concept of code smell, terrible decisions or choices on programming activities. Generally, these smells have detrimental effects on the comprehensibility of source code and thus, pose a threat to the maintainability during the software update. They have proposed 22 kinds of code smells, and presented detailed descriptions, features, as well as the refactoring strategies for each smell, respectively. Over recent years, code smell has highly attracted the attention of academy and industry, because previous studies found that smells or anti-patterns will bring more change- and fault-proneness to the affected source code [4–6].

Among the 22 smells, refused bequest is considered as one of the most harmful smells with high diffuseness [5]. This smell is introduced to indicate such subclasses that only partially use the functions or properties inherited from the parent classes [3]. However, it is rarely taken into account with only limited supports for the identification of refused bequest in previous works [7].

So far, two kinds of strategies have been exploited to tackle the problem of smell detection, including rule-based approaches [8–10] and machine-learning-based approaches [11, 12]. The rule-based detectors rely mostly on the selected or designed metrics, meaning that they require the specification of thresholds to distinguish smelly and non-smelly instances [13]. In contrast, the machine learning techniques are based on massive metrics rather than threshold of metrics. Ilyas et al. [7] conduct a systematic review and meta-analysis by summarizing the studies on code smell detection models. In their analysis result, some machine learning models work well on smell identification. However, there are still some limitations and room for the improvement of machine learning techniques in this field.

To some extent, the existing approaches could provide rational performance on the detection for some of code smells, there are two major challenges for the identification of the refused bequest smell:

- **Lack of dataset**. The rare concern of refused bequest is due to the fact that there is a lack of dataset. When creating a manually labeling dataset, it is required to consider polymorphism that is widely exploited in objective-oriented languages, which makes the labeling process tough and time-consuming. Liu et al. [14] claim that it is tedious to label the code smell instances manually, especially when smells involve more than one file. When analyzing refused bequest smell, most of the previous studies create the dataset by applying the existing detection tools and then manually evaluating the samples identified by detectors. While in this process, some real smelly instances might be misclassified as non-smelly instances, which means real-world smelly instances might

be missed [7]. Landfill [15] is a manually labeled dataset that is publicly available, while it only contains limited kinds of smells, without refused bequest.
- **Complicated rules**. The existing detection strategies for refused bequest rely mostly on rule-based techniques. However, it is also difficult to design the detection rules for refused bequest compared with other popular smells (i.e., god class, feature envy, long method). The number of overridden methods can be regarded as one of the principles to identify refused bequest, which is widely used in previous work. However, it is difficult to determine the threshold because of the large differences between projects. Besides, the identification strategy based on the overridden number might be intervened by template design pattern.

To address these challenges, in this paper, we propose a framework, SEADART (Smell gEneration assisted AdaBoost-based Detection Algorithm for Refused bequesT) that is able to leverage a set of synthetic smelly instances, and effectively identify the refused bequest smell. First, to tackle the challenge of lack of dataset, we propose a refused bequest smell generation technique from the projects with high quality. To generate positive instances with the smell, we deliberately create unreasonable inheritance relationships. After this operation, we could label the modified source files as smelly samples. Second, to tackle the complicated rules challenge, we adopt Adaptive Boosting (AdaBoost), as the smell detection model that is able to learn by examples rather than designing the detection rules. Palomba et al. [7] suggest exploiting the ensemble techniques to enhance the performance of code smell prediction models. Besides, AdaBoost could combine numerous typical machine learning algorithms ('weak learners') to improve the performance. Besides, using simple classifiers as weak learners can effectively avoid the over-fitting problem.

To evaluate the proposed approach, we first assess the effectiveness of the generated dataset by manually checking the reliability of the smelly instances. Second, we evaluate the performance of the AdaBoost model on the refused bequest smell detection. Experimental results demonstrate that AdaBoost significantly outperforms the other comparative machine learning models most widely used in previous studies. Finally, we examine the performance of AdaBoost on a real-world dataset, and the results show that the proposed model trained with the generated dataset is able to identify 78% refused bequest smell existing in the projects, performing considerably better than the state-of-the-art.

The paper makes the following contributions:

- We propose an ensemble-learning-based model to detect refused bequest, which features a set of synthetic dataset with refused bequest smell. To the best of our knowledge, this study is the first to cope with the lack of dataset of refused bequest smell.
- Evaluation results on the proposed approach show that the generated dataset is adequately effective for the training of the refused bequest detection model. The proposed AdaBoost model can significantly outperform the existing tools.
- An analysis of the metric importance for the task of the refused bequest smell detection is conducted, which gains further insights into the detection task for the refused bequest smell.

The rest of the paper is organized as follows. We first provide the background with a motivating example in Section II. The proposed approach is discussed in Section III. Section IV presents the evaluation result of our approach. The threats to validity and related work are described in Section V and VI. Finally, Section VII makes conclusions.

II. Background and Smell Example

In this section, we briefly describe the refused bequest smell and the corresponding refactoring method. Then, we provide an example to illustrate the reason why it is difficult to perform manual labeling and design detection rules.

A. *Refused Bequest*

Refused bequest is proposed to indicate such subclasses that only partially use the functions or properties inherited from the parent [3]. There are two circumstances involved in this smell and for each case, there are corresponding refactoring resolutions. On the one hand, if the subclass is created to reuse a bit of behavior inherited from the parent, the inheritance is appropriate. This symptom can be eliminated by creating a new sibling to push all the unused methods and fields to the sibling, following the traditional advice to let the parent only hold what is common [3]. However, in practice, there is no need to abide by the conventional rules strictly, which means that in this case, the smell can be ignored to some extent. On the other hand, if the subclass aims at reusing the behavior rather than supporting the superclass, the smell is much stronger. In this case, the symptom could not be simply addressed by applying the previous refactoring approach, and has a detrimental effect on the quality of source code [16]. To eliminate the smell, the inheritance relationship is required to be removed and replaced with delegation, keeping the functions of original projects unchanged.

B. *Motivating Example*

In this subsection, we present a typical refused bequest example to illustrate the reason why it is difficult to identify.

In Figure 1, there are four classes in this example, named Animal, Bird, Alpha, and Client, respectively. We assume that there is an inheritance hierarchy in which Animal is designed as the superclass of Bird and Alpha. The class Client can get access to the two subclasses. Obviously, the inheritance relationship between Animal and Bird is rational because subclass Bird specializes or overrides the function, move() and the attribute, "legs" provided by the super and supports other behaviors (i.e., breathe(), sleep()).

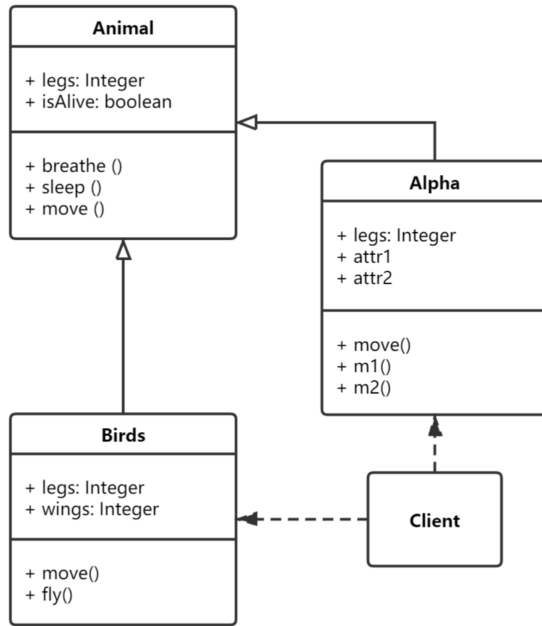

Fig. 1. Refused Bequest Example

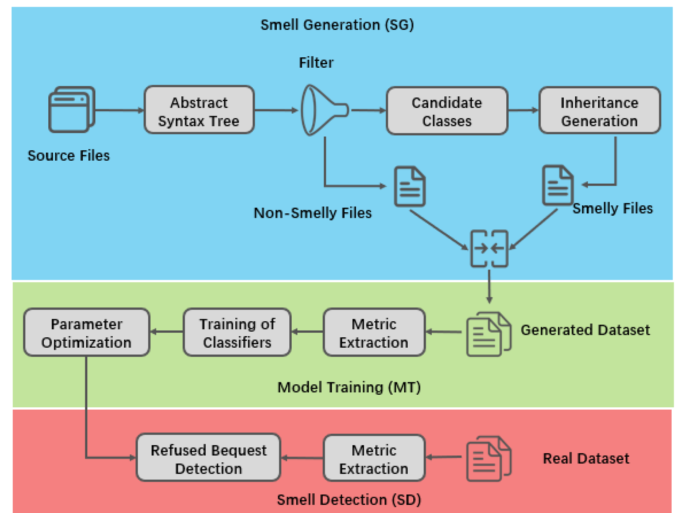

Fig. 2. The SEADART Framework

However, in the real world, the situation is much more complicated, as what is illustrated by the case between Animal and Alpha. It is difficult or unreasonable to determine the relationship between the two classes by the textual information, class and method name. Then, by analyzing the content and the structure of the code fragment, two main troubles appear.

Firstly, it is not enough to focus only on single class hierarchy itself, because polymorphism is widely exploited in object-oriented languages. Conducting an in-depth analysis of the hierarchy's clients [17] can confirm whether the inherited methods breathe() and sleep() are actually accepted by Alpha or not. We next focus on the class named Client, which is a simple example of the classes that contain the object of subclasses, Alpha, and Bird.

From Figure 1, we observe that if all of the implicitly inherited methods breathe() and sleep() are invoked by subclass Alpha, we could practically conclude that the inheritance relationship is rational without Refused bequest smell. The phenomenon means that if we can find one client in which the subclass is managing to use the features from its parent, the subclass could be removed from the suspect refused bequest list. Practically, the implicit functions rarely appear in one client simultaneously, and the candidate clients might be scattered in the whole project, leading to the complexity of identification. Another problem is that we could not absolutely infer the existence of refused bequest when failing to find out any invocation instance of the implicit methods in the project, as some programs are designed to provide functions for other projects (e.g., JUnit). Therefore, in order to avoid false-positive smell instances, there is a need to understand the real intention of source code to analyze the inheritance relationship. Therefore, it is tedious and time-consuming to perform manual labeling.

Furthermore, the number of overridden methods can be regarded as one of the principles to identify refused bequest, which is widely used in previous work. However, this strategy leads to bias that researchers made different thresholds of metric rule for the detection of refused bequest, causing the scarce agreement between different detectors or approaches [18, 19]. Another example is that the template design pattern is widely used in open-source software frameworks. In such pattern, the superclass is designed as the skeleton and provides some functions for its subclasses to override. Therefore, the identification strategy based on the number of overridden methods might misclassify the subclasses which exploit template pattern as refused bequest smell instances.

III. Our Approach

In this section, we discuss the design of the proposed framework. As illustrated in Figure 2, the framework can be divided into three main components, Smell Generation, Model Training and Smell Detection. For each component, we shall present the details and the implementation.

A. Smell Generation Component

The upper part of Figure 2 depicts the smell generation component, which aims at coping with the problem of lacking dataset by creating refused bequest instances according to its refactoring technique. The motivation behind the component is to deliberately mutate the high-quality source code by injecting code smell. In this study, we select the original projects with high quality and assume that there is no such smell in these projects. Therefore, based on this assumption, we can generate many refused bequest smelly instances by deliberately creating unreasonable inheritance relationships. Table I presents the

TABLE I
APPLICATIONS FOR MODEL TRAINING

Applications	Description	Version	NOC	NOM	KLOCs
Weka	Machine Learning Algorithms	3.9.0	1348	20,182	444
FreePlane	Knowledge Management	1.3.12	424	6,938	124
Areca	Document Backup	7.4.7	473	5,055	88
JExcelAPI	Excel API	2.6.12	424	3,118	90

statistics of the selected systems, i.e., the version, number of classes (NOC), number of methods (NOM), and lines of the source code (LOC).

First, given the source code of a project, we adopt the Eclipse JDT APIs to parse the code into an abstract syntax tree (AST). After building the AST of source files, we perform the candidate class selection through a filter. The candidates are those classes that can be infected with the smell with keeping the original function unchanged. The detailed process is shown in Algorithm 1. The filter takes AST as the input, and performs analysis on this project, in order to select the classes without superclass as candidates. Besides, abstract classes and interfaces are required to be removed from the candidate list, because these classes are impossible to be designed as subclasses.

As for the inheritance generation process, it is necessary to follow some principles. This is due to the fact that randomly choosing the parents for the candidates is irrational, resulting in the relatively large differences between the modified files and the practical refused bequest smells. As mentioned in section II, for each smell, there are corresponding refactoring strategies to eliminate it. Thus, we conduct an in-depth analysis of the refactoring approach of refused bequest, Replacing Inheritance with Delegation.

Algorithm 2 illustrates the specific strategy for the inheritance creation of those candidates. To select the ideal class to be extended for each candidate, we focus only on the classes defined in the project itself without considering the ones from the libraries or Jars. With this option, the generated inheritance hierarchy will appear in the same project. Besides, we should remove the classes that are defined as abstract or interface from the alternative parent list, since the refused bequest smell is rarely found between abstract or interface and their children. In particular, with all these optional parents, we follow the inspiration from the refactoring strategy to look for the delegation pattern existing in one Java source file and replace it with the inheritance relationship to create the refused bequest smell. Specifically, due to the wide distribution of delegation pattern, if there exists more than one such pattern, we choose the one that most intimates with the candidate as the superclass. In this paper, the level of intimacy is determined by how many times the class is invoked by our candidate.

Algorithm 1 Filter

1: **Input**: AST representation of the source code
2: **Output**: Candidate classes list
3: // Candidate classes selection
4: $candidate_list \leftarrow 0$
5: **for** each class c in the source code **do**
6:     **if** c is neither abstract nor interface with no parent **then**
7:         $candidate_list \leftarrow candidate_list \cup \{c\}$
8:     **end if**
9: **end for**
10: **Return**: $candidate_list.$

After confirming the parent of one candidate, the inheritance relationship could be established with the keyword “extends”. Considering the number of overridden methods that indicate the existence of refused bequest to some extent, we generate as many “overridden method” as possible in the candidate, to bring it closer to the real smell. More specifically, the operation could ensure that all of the generated overridden methods are not invoked by other classes in this project, which implies that these inherited methods are refused by the candidates.

Finally, it is required to deal with the problems caused by creating inheritance, to guarantee that the functionality of projects remains the same with no errors (i.e., name conflict, visibility of methods and attributes). After all the above processes, the refused bequest dataset has been generated with modified candidates as positive training items (smelly instances), and others as negative items (non-smelly instances).

B. Model Training Component

As mentioned in Section II, the metric-based detection approach might lead to bias with the scarce agreement between researchers and it is relatively difficult to design the heuristics rule. In order to avoid manually designed rules, in this paper, we choose the machine-learning-based approach to perform the refused bequest detection, which enables us to learn by examples instead of exploring the threshold of metrics.

After the smell generation phase, we can obtain a sufficiently large dataset to train the machine learning model. In this detection task, binary classification is applied to predict whether a given class is smelly or non-smelly. The dependent variable of our model is the smelliness of class, which can

TABLE II
METRIC DEFINITION

Category	Name	Definition	Category	Name	Definition
Basic	BUR	Usage ratio	Complexity	AMW	Average methods weight
	GEEDY	Raised exceptions		CC	Cyclomatic complexity
	PNAS	Public number of attributes		NOEU	Number of external variables
Size	LOC	Lines of code in the class		NOLV	Number of local variables
	LOCC	Lines of code classes		NOP	Number of parents in method
	NOM	Number of methods		WMC	Weight methods count
Cohesion	ALD	Access of local data		WOC	Weight of class
	TCC	Tight class cohesion	Inheritance	DIT	Depth of inheritance hierarchy
Coupling	ATFD	Access to foreign data		NOA	Number of ancestors
	CM	Changing methods		NOD	Number of descendants
	DAC	Data abstraction coupling	Encapsulation	LAA	Locality of attribute accesses
	FANOUT	Number of classes referenced		NOAM	Number of added methods
	FDP	Foreign data providers		NOPA	Number of public attributes

Algorithm 2 Inheritance Generation

1: **Input**: Candidate classes list $candidate_list$
2: **Output**: Smelly instances list
3: // Inheritance relationship generation
4: $parent_list \leftarrow 0$
5: **for** each class c in $candidate_list$ **do**
6:     **for** each class p in in source code **do**
7:         **if** p is defined in this porject and invoked by c **then**
8:             $parent_list \leftarrow parent_list \cup \{p\}$
9:         **end if**
10:     **end for**
11:     $intimate_index \leftarrow 0$
12:     $parent_class$
13:     **for** each class p in in $parent_list$ **do**
14:         Calculate the times of invocation w
15:         **if** $intamate_index \leq w$ **then**
16:             $intamate_index \leftarrow w$
17:             $parent_class \leftarrow p$
18:         **end if**
19:         Create inheritance relationship between $parent$ and c
20:         Eliminate the effects caused by new inheritance
21:     **end for**
22: **end for**
23: **Return**: $candidate_list.$

be labeled according to the result of the smell generation. As for the independent variables, we choose the widely used structural-based metric that plays a significant role to enhance the prediction performance.

In order to calculate these metrics, Iplasma [10] and Together[1] are employed in this study. We choose 26 well-known code metrics that are widely used in previous studies [11, 20, 21], covering different aspects of code, i.e., Basic, Size, Complexity, Cohesion, Coupling, Inheritance, and Encapsulation. The chosen metrics and their descriptions are illustrated in Table II.

After constructing the dataset with labels and metrics as features, we perform data preprocessing to deal with the missing values and duplicate data. For missing values, we select the average value on the feature to fill the blank. Meanwhile, duplicate data with identical class names are removed from the dataset.

As for the detection model, we apply an AdaBoost based classifier, due to its promising potential in enhancing the performance of code smell prediction models [7]. AdaBoost [22], short for Adaptive Boosting, is one of the most well-known algorithms in the boosting family. The algorithm trains models sequentially, with getting a new trained model at each round. For each round, the misidentified instances are recorded with weight increased in the next round. Based on this feature, we can select numerous typical learning algorithms as 'weak learners', including decision tree and support vector machine (SVM) to improve the performance. The ensemble process is established by iteratively adding models, and the final output of the boosted learner is a weighted sum that is combined by the output of several weak learners. By applying the AdaBoost algorithm on the detection, there is less need to worry about the over-fitting problem than other machine learning algorithms. Because using simple classifiers as weak

[1]http://www.microfocus.com/en-us/products/together

learners can effectively avoid over-fitting.

Finally, finding optimal parameters plays an important role in model training, which might have a significant impact on the performance of the model. For AdaBoost, the most important parameters are the learning rate and the number of weak learners. The Grid-search algorithm [23] is able to find the near-optimal value of parameters by exploring the parameter space. By performing cross validation, Grid-search can automatically optimize the selection of all possible combinations of parameters. And the whole parameter optimization process can be accomplished in less than five minutes.

C. Smell Detection Component

In order to further verify the availability of the generated dataset and the trained model, we perform the refused bequest detection on real-world projects. We choose a small, manually labeled dataset provided by the empirical study [5], in which there are 395 releases of 30 open source systems to investigate the density and harmfulness of 13 code smells including refused bequest. Therefore, among these projects, we only select the ones with more refused bequest smell instances to avoid the randomness of the detection result, which can ensure the reliability of the smell detection experiment. The detailed descriptions of selected systems are illustrated in Table III.

With these labeled datasets, we need to perform the metric extraction again and keep the selected metrics the same as the ones used in the generated dataset. In addition, we perform the same data preprocessing strategy so that the detection model can work normally. The real-world dataset is then fed into the trained model for smell detection. Palomba et al. [7] suggest that focusing on the cross-project code smell detection represents the right choice because the smell detection process are generally performed on completely unknown projects. Hence, for the trained model, we select the merged dataset from the four projects in the smell generation component, in order to eliminate the differences between different software.

IV. Evaluation

In this section, we intend to evaluate the proposed approach, by investigating the result of the smell generation approach on four open-source projects, as well as the performance of proposed detection model on five real-world applications respectively.

A. Research Questions

This study concentrates on the following research questions (RQs):

- RQ1: Does the smell generation method create the instances with real refused bequest smell?
- RQ2: Is AdaBoost able to outperform other machine learning techniques for the refused bequest detection?
- RQ3: Does the AdaBoost model trained with the generated dataset work well on real-world projects? How does the AdaBoost model perform when compared with existing rule-based tools for the detection of the refused bequest smell?
- RQ4: What are the most important metrics that indicate the existence of refused bequest?

Among these RQs, RQ1 evaluates the reliability of the refused bequest smell dataset generated by the designed approach. To answer this question, the generation result is manually checked by five postgraduate students. All these students major in software engineering and have internship experience with Java development. The manual checking process is conducted independently, and the participants exchange the examined files to cross-check. Finally, a group discussion is carried out to eliminate inconsistency.

RQ2 investigates the performance of AdaBoost in detecting the refused bequest smell compared with other machine learning models. There are some studies that have employed machine learning techniques to identify other smells with high accuracy. Thus, we choose the top two widely used models in existing works to conduct a comparison, according to the statistic [7].

RQ3 concentrates on the effectiveness of the trained model in the real world. Although the generated datasets have been manually evaluated, the smelly instances are not from real-world applications. Hence, it is necessary to evaluate the performance of the model again on real-world dataset. Answering this question would verify the availability of our model for the refused bequest detection, and further confirm the usability of the generated smell dataset by the proposed approach. RQ3 also concerns the comparison with existing rule-based tools. The comparative approaches considered in our experiment are two state-of-the-art tools, which are able to report the refused bequest smell in Java code: Iplasma [10] and Decor [8]. The two tools are chosen due to the following reasons. First, among these code smell detection tools, there are few designed for the refused bequest identification. Iplasma has been widely applied as a benchmark for code smell detection algorithms comparison [11, 24]. Besides, these tools are publicly available, which means it enables other researchers to reproduce our experiment to make the validation and conduct further study.

RQ4 aims at discovering the most important metrics for the identification of the refused bequest smell. Based on our trained AdaBoost model, we can obtain the correlation coefficient of metrics, in order to assess the importance of metrics, providing a reference for the researchers who aim at the metric-based detection techniques.

B. Subject Selection

In this study, we evaluate the proposed smell generation approach on four open-source applications presented in Table I. Since the proposed approach is accomplished by applying JDT tools provided by Eclipse, the selected projects are Java applications only. The columns report the name and the

TABLE III
APPLICATIONS FOR VALIDATION

Applications	Description	Version	NOC	NOM	KLOCs	Number of Smell
Derby	Relational Database Management System	3.9.0	1,929	28,119	734	17
Eclipse	Integrated Development Environment	3.6.0	1,181	18,234	441	10
Xecres	XML Parser	1.2.0	471	7,342	201	3
Pig	Large Dataset Analyzer	0.8.0	441	7,619	184	3
Hsqldb	HyperSQL Database Engine	2.2.8	513	8,808	260	12

description of the projects, the version, the number of classes, the number of methods, and the kLOCs, respectively.

These applications are chosen due to the following reasons. First, all of them are publicly available. Second, all of these projects are famous and considered to be of high quality [14]. As mentioned in Section III, the smell generation approach is based on the assumption that there exists no smell involved in the original projects.

For the smell detection experiment, we choose another group of applications, which are presented in Table III. These applications are filtered from the empirical study [5], which aims at investigating the density and harmfulness of 13 code smells including refused bequest, with providing a small dataset for each smell. Because there is little refused bequest smell in some applications and thus, we only select the ones with more refused bequest instances as our validation dataset to avoid the randomness of the detection result, which can ensure the reliability of the smell detection experiment. Among the five applications, Derby and Eclipse are large projects with more than 1,000 classes involved and the other three are relatively small.

C. Evaluation Metrics

In order to measure the effectiveness of the trained machine learning models and detection tools, we calculate the precision, recall, and F1-score that are widely used to evaluate the performance of algorithms.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \tag{1}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \tag{2}$$

According to the formula, Precision represents of all the predicted positive instances, the percentage of true positive. While Recall reports of all the actually positive instances, the percentage of correctly predicted as positive. Precision and Recall can account for the performance of detection results in different aspects. However, it is common that during model training and detection task, either precision is high and recall is low or vice versa, which brings difficulties to evaluate the model with two metrics.

TABLE IV
MANUALLY EVALUATED GENERATED SMELLY INSTANCES

Applications	Number/Percentage of Smell on Generated Dataset	Correct	Incorrect
Weka	131/8.43%	129	2
FreePlane	85/9.72%	82	3
Areca	47/8.62%	45	2
JExcelAPI	35/7.96%	35	0
Total	298/8.76%	291	7

F1-score is defined as the harmonic mean of Precision and Recall. Compared with balancing two separate metric together, F1 score is more convenient to work with and it is represented by the following formula:

$$F1\text{-}Score = 2 \times \frac{Precsion \times Recall}{Precision + Recall} \tag{3}$$

D. RQ1: Effectiveness of Generated Dataset

To investigate the effectiveness of generated dataset, the generation result is manually checked by five postgraduate students who are majored in software engineering and experienced in Java language. The manually evaluated results are presented in Table IV. The first column presents the name of applications and the second one presents the number and the percentage of generated smelly instances for each project. Columns 3-4 illustrates the number of correct and incorrect smelly instances, respectively.

As the table shows, the percentage of smelly instances in each project is very small. However, the density of refused bequest in our study is slightly higher than the real-world circumstance, because the smelly instances are generated deliberately, in order to create a sufficiently large dataset to train the machine learning model.

TABLE V
EVALUATION RESULTS ON PROPOSED DETECTION MODEL

Applications	AdaBoost			Decision Tree			SVM		
	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
Weka	0.961	0.854	0.901	0.787	0.800	0.788	0.758	0.738	0.740
FreePlane	0.831	0.822	0.817	0.729	0.711	0.699	0.752	0.544	0.602
Areca	0.697	0.820	0.732	0.575	0.600	0.579	0.563	0.720	0.577
JExcelAPI	0.915	0.950	0.927	0.795	0.825	0.800	0.950	0.825	0.869
Average	0.851	0.861	0.844	0.721	0.734	0.716	0.756	0.706	0.697
Merged	0.826	0.757	0.787	0.689	0.700	0.688	0.613	0.643	0.643

As mentioned in Section I, we assume that the selected projects are with high quality, which means there is no refused bequest smell in the original source code. Therefore, we only manually evaluate the generated smell items, with the evaluation results presented in columns 3-4. The Evaluation results suggest that the generation approach is able to produce reliable refused bequest smells. For each project, there are only no more than three incorrect smells. These instances are recognized as incorrect smells because it is difficult to estimate whether they are smelly or not when reading the source code.

E. RQ2: Performance of AdaBoost

To answer this RQ, we first compare the AdaBoost model with other machine-learning-based techniques, including decision tree and SVM. The comparison results are presented in Table V. The first column indicates the employed applications, datasets for model training and detection. The rest columns present the evaluation metrics on different machine learning models. Note that the result of the last row is evaluated by merging the dataset and then running models on it rather than simply calculating the average performance on the above applications.

From Table V we can obtain the following observations:

First, AdaBoost performs dramatically better than the other machine learning models in terms of F1-score. Its average F1-score achieves 0.84, while the average F1-score of decision tree and SVM is only 0.71 and 0.69, respectively. Besides, AdaBoost is able to recognize most of the refused bequest smells in these projects because the average recall gets to 0.86, which is 0.12 better than decision tree and 0.15 better than SVM. Apart from that, the average precision (0.85) of AdaBoost significantly outperform decision tree (0.72) and SVM (0.75) as well.

Among these four projects, the performance of smell prediction on Weka is the best, because there are more smelly instances generated by our approach, which enables the model to learn more smell-specific features from examples. The results also account for the importance of the annotated smell dataset for the smell identification when using machine-learning-based techniques.

Although on the merged dataset, the trained models fail to perform as well as on a single project, the result is in general acceptable, and the slight decline of the performance on the merged dataset is rational because of the large differences between projects. Also, it is necessary to build the model with robustness, in order to perform smell detection on other unknown projects, as what is suggested by Palomba et al. [7], paying more attention to the cross-project code smell prediction.

From the above analysis on the detection result, we conclude that AdaBoost significantly outperforms the other machine learning techniques that are most frequently applied in previous studies for code smell detection.

F. RQ3: Performance on real-world projects

In order to evaluate the performance of AdaBoost model trained by our generated dataset, we perform the smell detection experiment on a real-world dataset with manual labels, which is provided by the existing empirical study [5]. The detection result is presented in Table VI. In the table, the first column illustrates the name of the real-world projects. And for each project, there are three other columns indicating the evaluation metrics, which are the same as the ones in the previous experiment to investigate the performance of the proposed model and the existing tools, respectively.

From Table VI, we can make the following observations:

- First, the proposed AdaBoost model trained by the generated dataset is able to detect the refused bequest smell on the real-world projects. On average, the proposed model successfully identifies about 0.78 of the smells existing in the project. Especially, all of the refused bequest smell contained in project Pig are recognized by our model. However, because the percentage of smelly instances is quite low, the precision of the model only achieves about 0.60 on average, while the F1-score is 0.67, which indicates the result is acceptable.

TABLE VI
EVALUATION RESULTS ON REAL-WORLD DATASET

Applications	AdaBoost			Iplasma			Decor		
	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
Derby	0.570	0.670	0.610	0.100	0.600	0.171	0.180	0.400	0.248
Eclipse	0.500	0.800	0.620	0.180	0.400	0.248	0.200	0.500	0.286
Xecres	0.670	0.710	0.690	0.180	0.400	0.248	0	0	0
Pig	0.600	1.000	0.750	0.170	0.330	0.224	0	0	0
Hsqldb	0.690	0.750	0.720	0.120	0.250	0.162	0.180	0.145	0.161
Average	0.606	0.786	0.678	0.150	0.396	0.211	0.112	0.209	0.139

- Second, the proposed AdaBoost model can also significantly outperform the state-of-the-art Iplasma and Decor on the real-world dataset in detecting the refused bequest code smell. More specifically, the proposed model improves the precision from 0.15 (Iplasma) and 0.12 (Decor) dramatically to 0.60. Besides, as for the recall metric, AdaBoost also performs much better than Iplasma and Decor, meaning that the proposed model is able to detect much more true positive smelly instances.

From the detection results, we can find that the existing tools perform poorly because of the following reasons. First, these detection tools are rule-based, and the thresholds of the metrics are designed according to the projects in previous studies. The large differences between projects might cause unsatisfactory performance. Second, over projects Xecres and Pig, the results of Decor are zero, mainly because there are only three smells existing in the projects, yet Decor fails to identify the smelly instances.

We conclude from these results that the proposed AdaBoost based detection model is able to identify refused bequest smells on the real-world projects with high recall and acceptable precision, and performs significantly better than the state-of-the-art tools.

G. RQ4: Importance of Metrics

In order to analyze the most important metrics, we compute the correlation coefficients for all selected metrics and list the top 10 metrics in Table VII. In the table, the first column shows the ranking of each metric, and columns 2-3 illustrate the name and the category of the metrics. The last column is the calculation result of the correlation coefficient for each metric.

From the statistical results, we can observe the following phenomena. First, there are two metrics belonging to the inheritance category in the top five importance metrics, which indicates that the proposed detection model is reasonable. However, only considering the two metrics relating to inheritance are not sufficient to identify the refused bequest smells. As mentioned in section II, it is unreasonable to detect the refused bequest smell based on the number of overridden methods only, although most of the previous works take such strategy. Apart from that, there are about half of the metrics are relating to the size or complexity of source code, meaning that the bigger class is prone to contain refused bequest smell.

TABLE VII
RESULTS OF TOP 10 IMPORTANT METRICS

Ranking	Metric	Category	Importance
1	NOM	Size	0.39
2	DIT	Inheritance	0.38
3	NOAM	Encapsulation	0.35
4	WMC	Complexity	0.32
5	NOA	Inheritance	0.32
6	DAC	Coupling	0.30
7	LOCC	Size	0.19
8	BUR	Basic	0.17
9	WOC	Complexity	0.15
10	TCC	Cohesion	0.13

V. THREATS TO VALIDITY

After analyzing the performance of our experiment and model, it is worth mentioning that there are two main points that may pose threats to the validity.

First, in this study, we make an assumption that the handled original projects are with high quality, meaning that there is no refused bequest smell in these applications. Only based on this assumption, the result of the smell generation approach is reliable. However, the assumption might be invalid and hence, the evaluation of the effectiveness of the generated dataset may be inaccurate. In order to minimize the threat, we choose the well-known applications whose quality has been confirmed

by Liu et al. [14]. Besides, we perform manual checking on the generated dataset before applying it to model training to ensure the trained detection model is of great performance.

Second, although the generated dataset is checked manually, the automatically generated smelly instances might be different from those introduced by developers in real-world projects. Consequently, the model trained with such dataset may not support the real-world applications. To mitigate this threat, we consider some small datasets which are manually labeled by Palomba et al. [5], and run our detection model on the real-world dataset to confirm the availability of the proposed model. Besides, during this process, the usability of the generation approach can be confirmed, which means the automatically generated dataset can be used to train the model for the refused bequest smell detection.

VI. Related Work

A. Refused Bequest

Beck and Fowler [3] introduce the concept of code smell to indicate such subclasses that use only some functions or properties inherited from its parent. To enhance the comprehensibility and maintainability of software, they present the corresponding strategy "Replace Inheritance with Delegation" to eliminate such smell.

Kegel and Steimann [25] perform an informal analysis of the refactoring approach and propose a tool that is able to implement such refactoring. Their analysis results suggest that the refactoring is not always possible and may not be effective as expected. Therefore, they investigate indispensable preconditions and implementable postconditions and then build a tool to achieve the refactoring.

However, the identification of the refused bequest smell is rarely concerned in this community. Marinescu [26] proposes a detection strategy by combining the code metrics and the definition of the threshold. This mechanism is employed by capturing deviations from good design heuristics. Iplasma [10] is a publicly available tool that is designed for quality analysis of object-oriented applications. It is able to detect the refused bequest smell by using metric-based rules. Moha et al. [8] design DECOR to define the rules of code smell detection by using Domain-Specific Language and they extend the tool to support for the refused bequest detection.

Unlike these detection approaches, in this work, we exploit machine-learning-based technique, which is largely metrics based learning by examples strategy to identify the refused bequest smell.

B. Machine Learning Based Code Smell Prediction

Because machine learning techniques have been successfully applied in many fields, a large number of studies employ machine learning models to identify code smells [7]. Maiga et al. [27] introduce SVM to detect blob, functional decomposition, spaghetti code, and swiss army knife anti-patterns, which can be considered as the smell existing in software design. Maiga and Ali [12] extend the previous work by considering the feedback of developers. Fontana et al. [20] perform 16 different machine learning algorithms on 74 systems for the detection of four code smells (data class, large class, feature envy, and long method), and they extend the work for the prediction of the severity of smells [28]. Liu et al. [14] employ deep learning to detect feature envy by combining the textual and structural information of the source code.

However, despite the wide use of machine learning for code smell prediction, some of the smells are rarely supported, especially refused bequest, which has been confirmed to be harmful to developers [5, 29]. Meanwhile, the ensemble techniques have not been considered to enhance the performance of the detection models. In this study, we exploit the Adaptive Boosting model to identify the refused bequest smell and train the model with the sufficiently large dataset, which is generated by the proposed generation strategy rather than relying on the detection results of the existing smell detectors.

VII. Conclusion

In this paper, we present a framework SEADART, which is able to create reliable smelly instances and effectively identify the refused bequest smell. To tackle the challenge of lack of dataset, we propose a refused bequest smell generation strategy. Besides, we apply the generated dataset to train an AdaBoost based detection model. To verify the usefulness of the proposed detection model and the generated dataset, we perform the refused bequest smell detection on the real-world dataset. The evaluation results suggest that the generated smelly instances are reliable, and the trained AdaBoost model performs better than two most frequently used machine learning models, and significantly outperforms the state-of-the-art. Finally, we analyze the most important metrics for the identification of the refused bequest smell, in order to provide a reference for the researchers who aim at the metric-based detection techniques. However, the smell generation and detection framework is designed for java program. Thus, it is not generalizable to other languages, which becomes a limitation of the proposed approach.

For the future work, we intend to explore new metrics rather than considering the structural metrics only, in order to further improve the performance of our detection model. Moreover, we are interested in extending the smell generation strategy to make the created instances closer to the real-world smells.

Acknowledgement

The authors would like to thank the reviewers for their great efforts.

References

[1] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] W. Cunningham, "The wycash portfolio management system," *OOPS Messenger*, vol. 4,

no. 2, pp. 29–30, 1993. [Online]. Available: https://doi.org/10.1145/157710.157715

[3] M. Fowler, *Refactoring - Improving the Design of Existing Code*, ser. Addison Wesley object technology series. Addison-Wesley, 1999. [Online]. Available: http://martinfowler.com/books/refactoring.html

[4] F. Khomh, M. D. Penta, Y. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012. [Online]. Available: https://doi.org/10.1007/s10664-011-9171-y

[5] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9535-z

[6] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Trans. Software Eng.*, vol. 43, no. 11, pp. 1063–1088, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2017.2653105

[7] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Inf. Softw. Technol.*, vol. 108, pp. 115–138, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2018.12.009

[8] N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 20–36, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2009.50

[9] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 347–367, 2009. [Online]. Available: https://doi.org/10.1109/TSE.2009.1

[10] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel, "iplasma: An integrated platform for quality assessment of object-oriented design," in *Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool volume, ICSM 2005, 25-30 September 2005, Budapest, Hungary*, 2005, pp. 77–80.

[11] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro, "Experience report: Evaluating the effectiveness of decision trees for detecting code smells," in *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*. IEEE Computer Society, 2015, pp. 261–269. [Online]. Available: https://doi.org/10.1109/ISSRE.2015.7381819

[12] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. Guéhéneuc, and E. Aïmeur, "SMURF: A svm-based incremental anti-pattern detection approach," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. IEEE Computer Society, 2012, pp. 466–475. [Online]. Available: https://doi.org/10.1109/WCRE.2012.56

[13] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, EASE 2016, Limerick, Ireland, June 01 - 03, 2016*, S. Beecham, B. A. Kitchenham, and S. G. MacDonell, Eds. ACM, 2016, pp. 18:1–18:12. [Online]. Available: https://doi.org/10.1145/2915970.2915984

[14] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 385–396. [Online]. Available: https://doi.org/10.1145/3238147.3238166

[15] F. Palomba, D. D. Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Landfill: An open dataset of code smells with public evaluation," in *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, M. D. Penta, M. Pinzger, and R. Robbes, Eds. IEEE Computer Society, 2015, pp. 482–485. [Online]. Available: https://doi.org/10.1109/MSR.2015.69

[16] D. Taibi, A. Janes, and V. Lenarduzzi, "How developers perceive smells in source code: A replicated study," *Inf. Softw. Technol.*, vol. 92, pp. 223–235, 2017. [Online]. Available: https://doi.org/10.1016/j.infsof.2017.08.008

[17] P. F. Mihancea, "Towards a client driven characterization of class hierarchies," in *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*. IEEE Computer Society, 2006, pp. 285–294. [Online]. Available: https://doi.org/10.1109/ICPC.2006.48

[18] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, "Antipattern and code smell false positives: Preliminary conceptualization and classification," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 2016, pp. 609–613. [Online]. Available: https://doi.org/10.1109/SANER.2016.84

[19] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *J. Object Technol.*, vol. 11, no. 2, pp. 5: 1–38, 2012. [Online]. Available: https://doi.org/10.5381/jot.2012.11.2.a5

[20] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9378-4

[21] D. D. Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. D. Lucia, "Detecting code smells using machine learning techniques: Are we there yet?" in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 612–621. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330266

[22] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," in *Computational Learning Theory, Second European Conference, EuroCOLT '95, Barcelona, Spain, March 13-15, 1995, Proceedings*, ser. Lecture Notes in Computer Science, P. M. B. Vitányi, Ed., vol. 904. Springer, 1995, pp. 23–37. [Online]. Available: https://doi.org/10.1007/3-540-59119-2_166

[23] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *CoRR*, vol. abs/1801.10270, 2018. [Online]. Available: http://arxiv.org/abs/1801.10270

[24] M. A. S. Bigonha, K. A. M. Ferreira, P. P. Souza, B. L. Sousa, M. Januário, and D. Lima, "The usefulness of software metric thresholds for detection of bad smells and fault prediction," *Inf. Softw. Technol.*, vol. 115, pp. 79–92, 2019. [Online]. Available: https://doi.org/10.1016/j.infsof.2019.08.005

[25] H. Kegel and F. Steimann, "Systematically refactoring inheritance to delegation in java," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 431–440.

[26] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 350–359.

[27] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, M. Goedicke, T. Menzies, and M. Saeki, Eds. ACM, 2012, pp. 278–281. [Online]. Available: https://doi.org/10.1145/2351676.2351723

[28] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowl. Based Syst.*, vol. 128, pp. 43–58, 2017. [Online]. Available: https://doi.org/10.1016/j.knosys.2017.04.014

[29] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia, "Do they really smell bad? A study on developers' perception of bad code smells," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 101–110. [Online]. Available: https://doi.org/10.1109/ICSME.2014.32