Recommending Clones for Refactoring Using Design, Context, and History

Wei Wang and Michael W. Godfrey
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{w65wang, migod}@uwaterloo.ca

***Abstract*—Developers know that copy-pasting code (aka *code cloning*) is often a convenient shortcut to achieving a design goal, albeit one that carries risks to the code quality over time. However, deciding which, if any, clones should be eliminated within an existing system is a daunting task. "Fixing" a clone usually means performing an invasive refactoring, and not all clones may be worth the effort, cost, and risk that such a change entails. Furthermore, sometimes cloning fulfils a useful design role, and should not be refactored at all [23]. And clone detection tools often return very large result sets, making it hard to choose which clones should be investigated and possibly removed.**

In this paper, we propose an automated approach to recommend clones for refactoring by training a decision tree-based classifier. We analyze more than 600 clone instances in three medium- to large-sized open source projects, and we collect features that are associated with the source code, the context, and the history of clone instances. Our approach achieves a precision of around 80% in recommending clone refactoring instances for each target system, and similarly good precision is achieved in cross-project evaluation. By recommending which clones are appropriate for refactoring, our approach allows for better resource allocation for refactoring itself after obtaining clone detection results, and can thus lead to improved clone management in practice.

I. Introduction

Software clones are code snippets that are strongly similar to each other. Usually, software clones come into existence when a developer decides to copy-paste a code snippet from elsewhere in the system that solves a problem that is highly similar to the one under consideration. Empirical studies show that software clones often comprise 10%–20% of total source code base for large software systems of many application domains, such as operating systems [42], development environments [37], and database systems [40]. Code duplication offers developers a shortcut to reuse an existing design. However, cloning carries the risk of reducing code readability or maintainability; sometimes, cloning may even introduce or propagate software bugs [20].

To address problems associated with software clones, researchers have devised many automated clone detection algorithms to efficiently analyze large software systems. Successful clone detectors include CCFinder [22], NiCad [38], and iClones [14]. All of these tools are able to identify code duplication with gaps in large software systems. In 2012, a mainstream development environment [8],[18] also integrated clone detector into its static analysis toolset, allowing millions of developers to integrate clone detection into their own software development process.

Despite the success of detecting code clones with high accuracy and scalability, managing the clones within ever-evolving software systems remains challenging. Researchers have investigated many approaches to aid clone management. One primary approach is to develop visualization techniques so that developers can inspect clones within a development environment [8], track clones along versions of software systems [10], [19], and view differences among clone siblings [30].

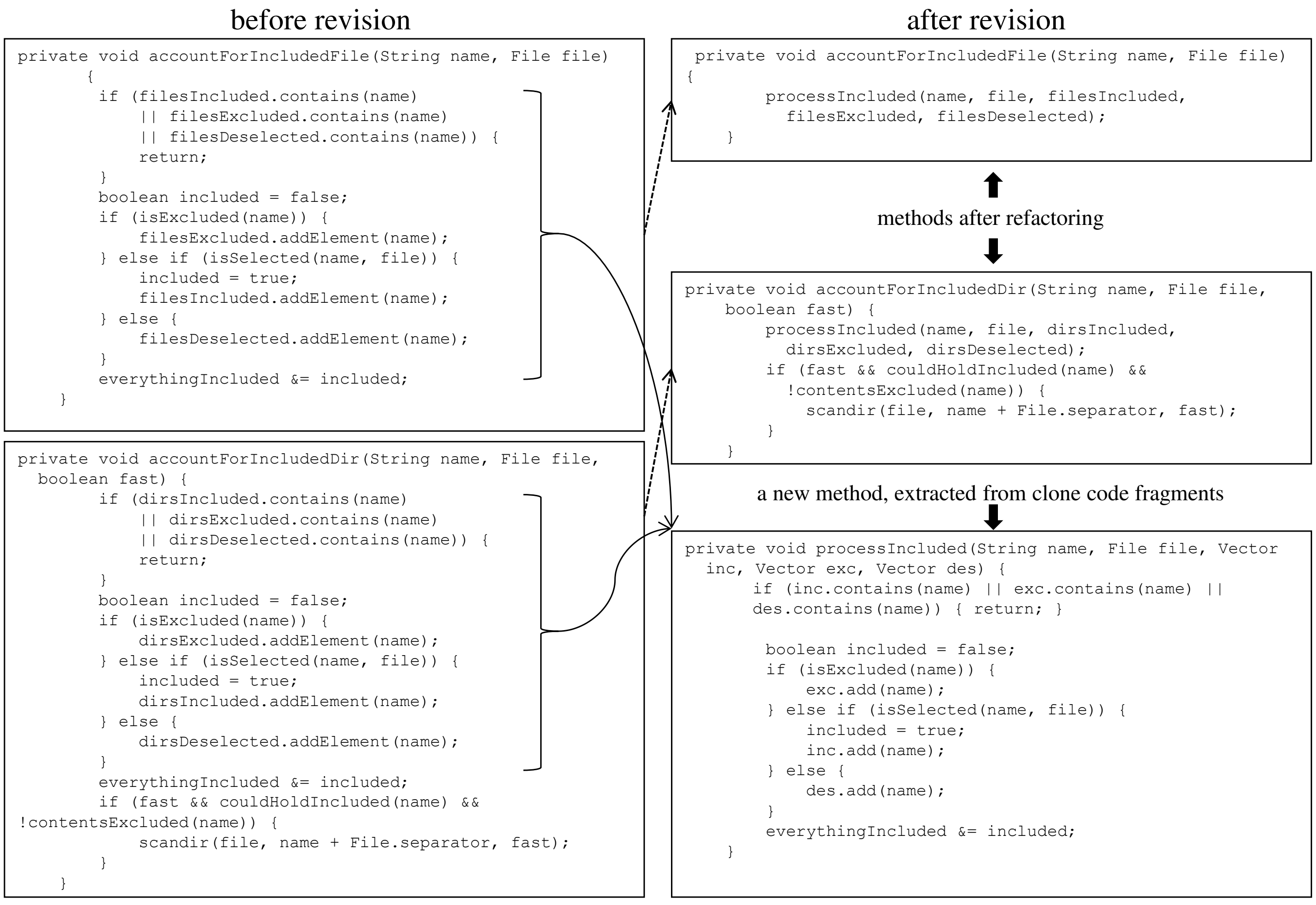
In clone management, if a developer attempts to "fix" a clone, refactoring is the primary option. *Clone refactoring* merges duplicated code snippets into a single place within the design of the system, while preserving the original functionality. Clone refactoring can reduce the risk of introducing or propagating bugs, and can also lead to long-term improvement of code maintainability and readability.

Concretely, our paper addresses this research question:

Assuming we have the results of a clone detection analysis on a software system, can we recommend clones that are appropriate for refactoring in terms of the likely benefits, costs, and risks?

To assess the appropriateness for refactoring for each clone, we use a machine learning approach to recommend clones for refactoring by learning from refactoring history of clones. More specifically, we collect clone refactoring instances from three target systems, then extract features that are related to the design, context and the history of cloned code. After collecting features from both "refactored" and "unrefactored" clones, we then consider clone refactoring recommendation as a classification problem.

The motivation to use a classifier to recommend which clones to refactor can be justified by experience and prior research. First, it is our own experience from both open source systems and proprietary systems that many features of clones — such as the location of clone siblings — can be important parameters when evaluating the cost-effectiveness of refactoring a clone. Second, prior research has confirmed the feasibility of our proposed approach in this paper. For instance, cloning patterns, first proposed by Kapser et al. [23], suggest that in some cases cloning is an ad-hoc solution to quickly reuse an existing design in a new context, while the best practice is a new abstraction that can serve both the existing and new contexts; but in other cases, clone instances are


before revision
after revision
methods after refactoring
a new method, extracted from clone code fragments

```
private void accountForIncludedFile(String name, File file)
       {
        if (filesIncluded.contains(name)
            || filesExcluded.contains(name)
            || filesDeselected.contains(name)) {
            return;
        }
        boolean included = false;
        if (isExcluded(name)) {
            filesExcluded.addElement(name);
        } else if (isSelected(name, file)) {
            included = true;
            filesIncluded.addElement(name);
        } else {
            filesDeselected.addElement(name);
        }
        everythingIncluded &= included;
    }
```

```
private void accountForIncludedDir(String name, File file,
  boolean fast) {
        if (dirsIncluded.contains(name)
            || dirsExcluded.contains(name)
            || dirsDeselected.contains(name)) {
            return;
        }
        boolean included = false;
        if (isExcluded(name)) {
            dirsExcluded.addElement(name);
        } else if (isSelected(name, file)) {
            included = true;
            dirsIncluded.addElement(name);
        } else {
            dirsDeselected.addElement(name);
        }
        everythingIncluded &= included;
        if (fast && couldHoldIncluded(name) &&
!contentsExcluded(name)) {
            scandir(file, name + File.separator, fast);
        }
    }
```

```
 private void accountForIncludedFile(String name, File file)
{
        processIncluded(name, file, filesIncluded,
          filesExcluded, filesDeselected);
    }
```

```
private void accountForIncludedDir(String name, File file,
    boolean fast) {
        processIncluded(name, file, dirsIncluded,
          dirsExcluded, dirsDeselected);
        if (fast && couldHoldIncluded(name) &&
          !contentsExcluded(name)) {
            scandir(file, name + File.separator, fast);
        }
    }
```

```
private void processIncluded(String name, File file, Vector
  inc, Vector exc, Vector des) {
       if (inc.contains(name) || exc.contains(name) ||
       des.contains(name)) { return; }

        boolean included = false;
        if (isExcluded(name)) {
            exc.add(name);
        } else if (isSelected(name, file)) {
            included = true;
            inc.add(name);
        } else {
            des.add(name);
        }
        everythingIncluded &= included;
    }
```

Fig. 1: Details of Refactoring of Clone Siblings in Revision *r436724* of Apache Ant.

created to evolve independently from each other, for purposes such as design experimentation. Following this observation, it is a reasonable assumption that certain features of cloned code snippets may be sufficient to identify clone patterns that are appropriate for refactoring. In addition to clone patterns, empirical studies by Göde and Koschke [13], [15] also indicate that only a small proportion of clones ever experience any kind of code-level changes. This observation implies that, in practice, an even smaller proportion of clones are refactored. In addition to clone patterns, our research can also be justified by the result from the work of Wang et al.[46], who find that certain features of clones can predict the likely change consistency (and therefore, the "harmfulness" of the clone, according to their definition) of a potential clone at the time of *introducing new clones*. Their research suggests that features of cloning may also predict the benefit, cost, and risks of refactoring *existing* clones.

To recommend which clones may be good choices for refactoring, we use a history of clone refactoring — as extracted from the version control histories of three open source systems — as our training data. After identifying clone refactoring instances by a combination of metrics, we extract features that relate to cloned code snippets, their context, and the cloning relation. We have found that our classifier recommends clone refactoring instances with high accuracy compared to our extracted ground truth. Additionally, cross-project testing indicates that a model trained from one software project can be applied to other software projects studied in this work with high accuracy.

This paper is organized as follows. In Section II, we conceptualize the problem of recommending clone refactoring by providing a concrete instance of clone refactoring found in Apache Ant. Following this, Section III introduces technical details of collecting clone refactoring instances, extracting features for classifier building, and classifier building. Section IV discusses the performance of our classifier, including the results of within-project validation, and cross-project validation, and the performance of subsets of features. We survey related work in Section VII, and summarize our contributions in Section VIII.

II. Motivation

A. Clone Refactoring in an Open Source Project

Refactoring of cloned code snippets has been observed in the evolution of many software systems [2],[45]. By refactoring, developers remove the duplication, primarily by extracting new methods (this is called "Extract Method" in Fowler's catalog [11]), or by creating a new method in the superclass ("Pull-up Method" in [11]). Clone refactoring can improve code maintainability, since developers no longer need to enforce change consistency among clone siblings.

As an example, let us consider an instance of clone refactoring in Apache Ant. In Revision *r436724* of Apache Ant (Figure 1), a developer wrote the following in the commit log: "*refactoring DirectoryScanner to reduce duplicated code, tests all pass*". Changes in source code confirm the refactoring of clone siblings in `DirectoryScanner`: clone siblings scattered across two methods are extracted into a new method called `processIncluded`, and the two methods invoke the new method to preserve the external behavior of the program.

Consider how many clones a developer might encounter before *r436724*. For the public release before *r436724*, our clone detector found 1,666 clone classes in Apache Ant comprising 4,801 code snippets. Even supported by a visualization tool, it is unrealistic for developers to decide manually whether each clone class is appropriate for refactoring. The impracticality of analyzing and managing a prohibitively large volume of clones has also been discussed by industrial developers [7], [6].

Code refactoring in general is considered by developers as a task with substantial cost and risks [26]. This observation is certainly valid for refactoring of clones. In Revision *r436724*, developers have to at least ensure the syntactic and semantic correctness of the refactoring, which entails comprehending the cloned code snippet and its surrounding code, the pre- and post-conditions of the cloned code snippets, and expected behavior of new extracted method. In addition to the costs and risks, a developer may also weigh the benefits of the refactoring. In *r436724*, the original cloned code snippet contains a sequence of conditional statements. Therefore refactoring clone siblings into a single method improves code readability and reduces long-term maintenance effort for the sequence of conditional statements.

We believe that developers' analysis of the cost, risks, and benefits of a possible clone refactoring is based upon features of clones, including features from the cloned code snippets, from their context, and from the cloning relation. Therefore, we have used a supervised machine learning approach to process features of clones, so that our approach can capture the essence of developers' analysis. More specifically, we train a decision tree-based classifier to learn from features from both refactored and unrefactored clone instances found in clone evolution history. Our approach is evaluated by ten-fold cross validation, and cross-project validation. Our approach frees developers from manually collecting and analyzing information for decision making, allowing developers to focus on those clones that appear to be most appropriate for refactoring.

III. Methodology

In this section, we describe the technical details of our approach. The overall workflow consists of five main steps: (1) clone detection; (2) identifying clone refactoring cases, along with clone instances that are not refactored; (3) extracting features from clone instances; (4) classifier training; and (5) performance assessment of classifier. We now explain each step in detail.

A. Definition and Terminology

We define a ***clone class*** C as a set consisting of two or more ***clone fragments***, and for any two of clone fragments of C, the two clone fragments must satisfy a "is-a-clone-of" definition to each other. In this paper, our definition of "is-a-clone-of" allows for minor differences in identifier, literals, types or white spaces (Type-2 Clones), our definition further allows for gaps among clone fragments (Type-3 clones). Details of this definition are provided by Bellon et al. [3].

We define ***clone refactoring*** as a change of code where at least two clone fragments of one clone class C are merged, and the external behavior of original code snippets is preserved.

It is worth noting that for a clone class C with more than two clone fragments, C is considered a clone refactoring instance if any two of the clone fragments are refactored. This definition reflects the fact that developers may be aware of only a subset of clone fragments of a clone class, especially since developers do not often use automated clone detectors.

B. Clone Detection

To detect clones in this study, we use iClones, a token-based clone detector of the Bauhaus Tool Suite.[1] We tune iClones to detect clones with gaps (i.e., Type-3 clones [3]), and set 20 tokens as the minimum threshold for iClones to consider a code snippet to be a clone snippet. We use 20 tokens as threshold because we find that it captures most of the clones that are refactored according to the commit logs of version control system in our pilot study for this paper. This setting is also used in a prior study [46]. In this paper, we exclude from analysis all testing programs within the target systems, as they often contain clones but are rarely refactored.

C. Identifying Refactoring of Clones

We chose three well-known open source systems as our targets: ArgoUML, Apache Ant, and Lucene. Details of our subject systems are listed in Table I. We choose these systems primarily because all three have been studied by prior research in cloning; also, these systems are fairly large and have an extensively documented evolutionary history, making them good targets for study.

We collected all available public releases of these systems, and identified refactoring instances between each pair of consecutive public releases. To identify clone refactoring instances in the selected target systems, we first generate candidate sets of clones based on a combination of metrics associated with refactoring, an approach similar to that of Demeyer et al. [9]. A clone class C is a candidate for manual processing if *any* of the following criteria are met:

1) at least two siblings of the clone class C disappear over consecutive versions;
2) for a method in which a cloned code snippet resides, its number of source line of code shrinks, and at least one new method invocation is added in that method; or

[1]http://www.bauhaus-stuttgart.de/bauhaus/index-english.html

TABLE I: Facts of Target Systems

Target System	Size (*Min-Max* LOC)	Duration (Month)	# Releases	Start Release	End Release	# Commits	# Identified Clone Refactoring Instances
ArgoUML	82,074 - 133,415	111	19	0.10.1	0.34	8,731	109
Apache Ant	8,920 - 106,225	126	27	1.1	1.9.3	12,960	87
Lucene	19,205 - 89,596	101	37	1.9.1	4.6.1	9,030	127

3) a new source class is created as a superclass of the class in which the original cloned code snippet once resides.

We choose these three metrics since they capture all cases of refactoring patterns that are related to clone refactoring. By using this metric-based approach, we select clone evolution candidates for manual inspection. According to Demeyer et al. [9], this approach can detect all of Fowler's refactoring patterns [11] that are applicable to clone refactoring, including "Extract Method", "Extract Superclass", "Pull-Up Method", "Replace Method with Method Object", and "Template Method". After generating the candidate sets, we manually inspect the evolution of each clone instance to determine whether the original functionality of the clone code snippets are preserved.

D. Alternative Approaches to Identify Clone Refactoring

In this paper, we use a metric-based approach to generate candidate sets of clone refactoring. Manual inspection is therefore required to further filter out false positives, i.e., clones that are not refactoring instances. Existing research suggests alternative approaches to identifying clone refactoring instances. However, in our studies we found that these alternative approaches may miss a significant proportion of clone refactoring instances. We now list these alternative approaches and explain why we have chosen not to use them.

- ***Natural Language Processing of Commit Logs***
  Developers may mention refactoring of clones in commit logs of version control systems. In our study, we have also tried to use keywords matching to identify source code commits with clone refactoring, a method used in other studies of clone evolution [13], [45]. However, when we tried it, we found that this approach missed the majority of clone refactorings in our sample; this is because developers often fail to provide textual descriptions that are precise and complete for each commit in commit logs. This phenomenon has also been observed by Parnin et al. [34] for code refactoring in general.
- ***Automated Refactoring Detection Tool***
  Prior research on clone refactoring [5] has used REF-FINDER [35] to identify clone refactoring instances. REF-FINDER is an automated tool that uses template logic rules to identify code refactoring patterns proposed by Fowler [11]. Unfortunately, after experimenting REF-FINDER on our target systems, we found that a substantial number of refactoring cases were not findable by the tool, such as clones that are refactored into nested method calls.

Other approaches to automatic identification of refactoring instances have been examined by Murphy-Hill et al. [32], including observing programmers' activities and mining logging data of developers' tools. Since neither of these approaches is feasible under the setting of our study, we do not discuss them further.

E. Collecting Clones With No Refactoring History

To train a classifier, we need a data corpus that includes both clone refactoring instances and clones without refactoring. To collect clones with no refactoring history ("unrefactored clones"), we select clones without any code reconstruction (including renaming identifiers) from the inception of clones to the last version we observe in this study. Given that unrefactored clones generally outnumber refactoring instances by several orders of magnitude, training all clones instances would introduce bias towards unrefactored clones when constructing the classifier. To solve this problem, we use random undersampling [16] to reduce the size of unrefactored clones. For each subject system, we randomly sample clones instances with no refactoring history at any version, so that we reduce the size of unrefactored clones equal to refactoring instances.

F. Feature Selection

To correctly recommend clone instances for refactoring, our classifier is expected to learn from features that can capture the benefit, cost, and risks that each refactoring entails. Therefore, we choose features from the following dimensions: (1) features associated with the cloning relationship; (2) features associated with cloned code snippets; and (3) features associated with the context of cloned code snippets. We offer full description of all features in Table II. Some of these features are used in prior research that train feature-based classifier for other purposes [43], [46].

In terms of feature extraction, Features 1, 3, 9, 10 derive from the clone detection results provided by ICLONES. For most of the remaining features in Table II, their extraction was effected by our implementation of PMD[2], a lightweight source code analysis tool. However, there are two features whose extraction could not be automated: Feature 11 (whether clone code snippet contains a complete control flow block) and Feature 12 (cyclomatic complexity of cloned code snippet). In these cases, we manually extract the values of these features. It is worth noting that features we present here are easy to extract. We have avoided some promising features, such as change frequency of cloned code snippets, as they are expensive to extract and often requires tools unavailable from an average developer.

G. Building Classifier For Recommending Clone Refactoring

In this paper, we choose to use a decision tree-based classifier C4.5 [36]. We choose C4.5 as it generates predictive

[2]http://pmd.sourceforge.net/

TABLE II: Feature Selection for Decision Tree Model Training

Type of Feature	Feature	Rationale
Cloning Relation	1) Number of clone fragments in a clone class	Ensuring consistency among many clone fragments requires increased maintenance efforts.
	2) Minimal Levenshtein distance among method names of clone fragments	Similar method names likely indicate similar design among clone fragments.
	3) Whether a clone class is a Type-3 Clone	Refactoring Type-3 clones may require more effort than refactoring Type-1 or Type-2 clones.
	4) Whether clone fragments are located in same file or in OO siblings	It is easier to apply "Pull-Up Method" refactoring for clones with this feature.
Context of Clone	5) Whether cloned code snippets immediately follow a control flow statement	Clones with this feature are more likely to indicate a standalone design.
	6) Lifetime (month) of a source file containing a clone code snippet	Cloning might be used as an ad-hoc solution to implement functions in new files, therefore refactoring may be desired.
	7) Lines of code of methods that contain the cloned code snippets	A long method may suggest increased complexity surrounding cloned code snippets, therefore refactoring might be difficult.
	8) Size of cloned code snippet out of size of a method	A larger size likely indicates a higher feasibility of applying "Pull-Up Method" refactoring, which requires less efforts compared to other refactoring methods.
Cloned Code Snippet	9) Size (LOC) of a cloned code snippet	A larger clone fragment may represent an increased benefit in maintainability.
	10) Size (# of tokens) of a cloned code snippet	Similar to Feature (9), but it ignores comments or empty lines.
	11) Whether a cloned code snippet contains a complete control flow block	A complete control flow block suggests a higher likelihood of extracting the clone siblings as a separate method.
	12) Cyclomatic Complexity of cloned code snippets	It indicates the complexity of understanding (thus refactoring) the cloned code snippets.
	13) Proportion of method invocation statements of a cloned code snippet	Refactoring of clones with method invocation statements may require more efforts, because of more problems such as method visibility, override, or exception handling.
	14) Proportion of arithmetic statements of a cloned code snippet	A high proportion of arithmetic statements implies that the design is essentially arithmetic, and therefore refactoring is likely necessary.
	15) Whether the clone snippet begins with a control flow statement	Beginning with a control flow statement may indicate the cloned code snippet represents a design that can be extracted into a separate method.

accuracy no inferior than other classification methods (such as Naive Bayes or SVM) in general. We use the implementation of C4.5 from the Weka toolkit, with the default settings (including pruning).

IV. Performance of Clone Refactoring Recommendation

A. *Performance of Within-Project Testing*

After labeling clone instances as "refactored" or "unrefactored", and extracting features from clone instances, we first train decision trees with clone instances for each subject system.

For a classifier that correctly recommends one clone refactoring instance as "recommended for refactoring", we denote this instance as $I_{r\to r}$. Classifying one clone refactoring instance as "not recommend for refactoring" is denoted as $I_{r\to n}$. Likewise, we have $I_{n\to n}$ instances (a correct classification that does not recommend "for refactoring") and $I_{n\to r}$ (an incorrect classification that falsely recommends for refactoring). We use precision, recall, and F-measure to evaluate the performance of a trained classifier for each subject system. These measures are defined as follows.

$$Precision_{refactoring} = \frac{I_{r\to r}}{I_{r\to r} + I_{n\to r}};$$

$$Recall_{refactoring} = \frac{I_{r\to r}}{I_{r\to r} + I_{r\to n}};$$

$$F-Measure_{refactoring} = \frac{2 * Precision_r * Recall_r}{Precision_r + Recall_r} \blacksquare$$

The definition for precision, recall, and F-measure for the unrefactored clone group are similar to that of the refactored clone group.

To evaluate within-project performance of our overall approach, we use ten-fold validation to test the classifier. We show the precision, recall, and the F-measure for both refactoring and non-refactoring group in Table III. For the refactored group, ten-fold validation generates precision from 78.3% to 87.9%; and the recall ranges from 74.7% to 91.3%, slightly lower than precision. The precision and recall for the unrefactored group are similar in the refactored group, with the precision from 77.3% to 90.8%. Given that the two groups in the training data are the same size, we can see that the classifier performs significantly better than random selection.

B. *Performance of Cross-Project Testing*

The performance of cross-project classification is important for both researchers and practitioners. For researchers, performance of cross-project recommendation can indicate the potential generalizability of our proposed approach. For practitioners, good performance of cross-project testing means

TABLE III: Precision and Recall of Within-Project Classifier Training Using C4.5

Project		Precision	Recall	F-Measure
ArgoUML	Refactored Group	0.808	0.771	0.789
	Unrefactored Group	0.908	0.871	0.889
Apache Ant	Refactored Group	0.783	0.747	0.765
	Unrefactored Group	0.773	0.806	0.789
Lucene	Refactored Group	0.879	0.913	0.896
	Unrefactored Group	0.908	0.871	0.889

TABLE IV: Precision of Recommending Clone Refactoring in Cross-Project Evaluation

Test System \ Model-Building System	ArgoUML	Apache Ant	Lucene
ArgoUML	X	0.752	0.806
Apache Ant	0.732	X	0.769
Lucene	0.885	0.842	X

TABLE V: Recall of Recommending Clone Refactoring in Cross-Project Evaluation

Test System \ Model-Building System	ArgoUML	Apache Ant	Lucene
ArgoUML	X	0.89	0.621
Apache Ant	0.598	X	0.761
Lucene	0.606	0.969	X

that a classifier trained on another project can be used with confidence on a new project.

In cross-project testing, we use the model trained on one subject system (the model-building system) and apply the model to other systems (the test system). We also use precision and recall of the classifier running on the test systems for performance evaluation. The precision and recall of cross-project evaluation is shown in Table IV, and Table V respectively. The highest precision (0.885) is generated when we apply the classification model obtained by learning from ArgoUML data on Lucene. From Table V, we can observe that the overall recall of cross-project testing is lower than results in the within-project setting. We further note that, since all of our training data is evenly split, both precision and recall of each cross-project testing is consistently superior than a random recommendation. For researchers, the results of cross-project testing indicates that the model trained by one single subject system is applicable to other systems. Moreover, by testing the model with data from other projects (unseen data during the training process), we reduce the possibility of model overfitting, a problem that affects how to evaluate the result in within project testing. Model overfitting is discussed in detail in Section V-B. For practitioners, the results of cross-project testing indicates a promising usability, as our approach can recommend clones that are appropriate for refactoring with the highest precision being 0.885, the lowest precision 0.732.

C. Classifier Training with Subsets of Features

After obtaining classification results from both within-project and cross-project settings, we are also interested in understanding the predictive power of each feature to the outcome of classification. Since it is widely agreed that features located near the root of the decision tree do not represent its predictive power [21], we instead select different subsets of features, and use each subset to train a new classifier. By doing so, we maintain the prediction accuracy by less use of import. Future research can also build upon our result by adding new features to our current approach. Similar to the setting of within-project training IV-A we also run ten-fold cross validation.

We select four subsets of features. Three subsets are directly from (see "*Type of Feature*" in Table II): **Cloning Relation**, **Context of Clone**, and **Cloned Code Snippet**. The fourth subset is **all features except for Feature 11 and 12**, since extracting features from the fourth subset can be fully automated, The results are shown in Figure 2 (for refactoring group), and in Figure 3 (for non-refactoring group). From both figures, we can observe that no single subset of features that consistently outperform other subsets in all three target systems. Features of cloned code snippets can yield the closest results when compared with the classifier that is trained with the entire set of features. We also note that features related to the context of clones can also generate results that are only slightly inferior to a classifier trained with all features. We also note that little performance improvement can be gained by learning from features on cloning-relations, such as Feature 3 (whether the clone class is a Type-3 clone), or Feature 4 (whether the clone fragments are in the same file or are OO-siblings). We also observe from the results that features from clone code snippets alone — independent of the type of clones or their context — can yield useful results to recommend clones for refactoring. The subset without Feature 11 or 12 has similar performance with the result using all features, suggesting the usefulness of our approach if developers only have time to extract features by fully automated tools.

V. Discussion

A. Cost of Constructing Classifiers

For developers, the practicality of our approach depends not only on the performance of the classifier but also on the cost of feature extraction. In our study, we deliberately choose features that can be obtained through lightweight source code analysis; some of the features do not even require that the subject systems to be compilable. Therefore, we consider that the deployment and feature extraction costs of our proposed approach are acceptable for a typical industrial software development setting.

Besides feature extraction, labeling clone instances with its refactoring history can be costly in practice. However, given the good performance of cross-project testing, developers can choose to apply a classifier that is trained on previous systems to a new system. This option is especially desirable when we are working on a new software system with a short evolution history.

B. Model Overfitting

Model overfitting refers to the possibility that by learning from the training data, our classification model incorporates unnecessary complexity to achieve high performance, thus generating spurious performance in terms of precision and

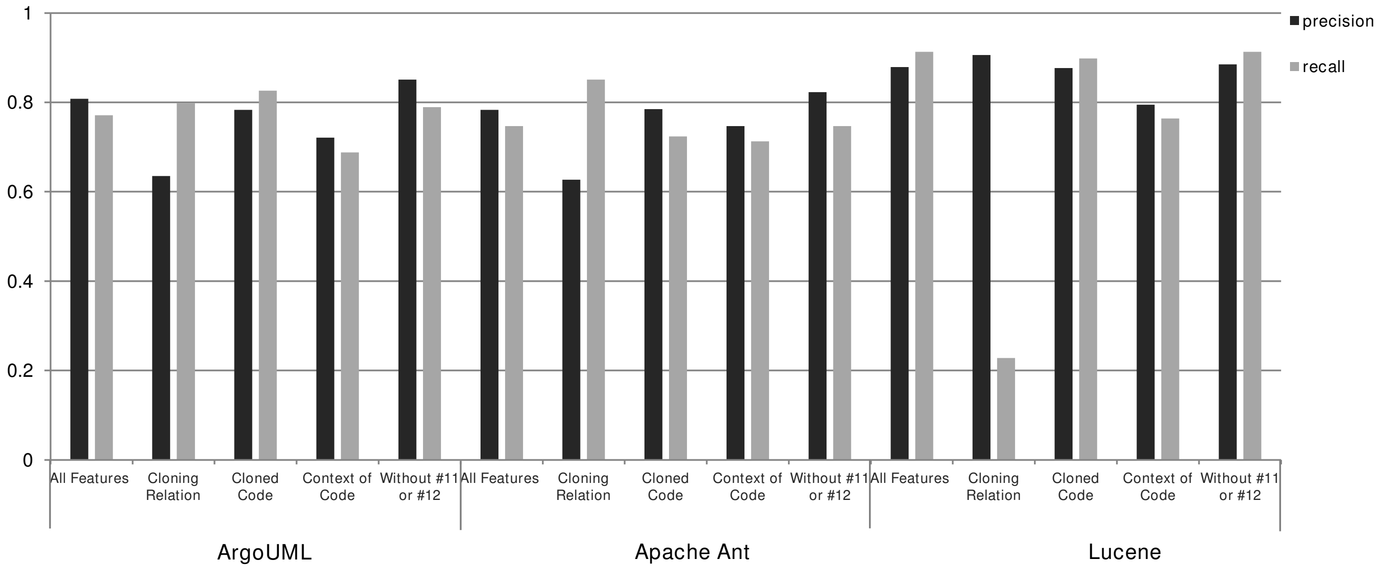


Fig. 2: Performance of using subsets for classification training for clone refactoring group.

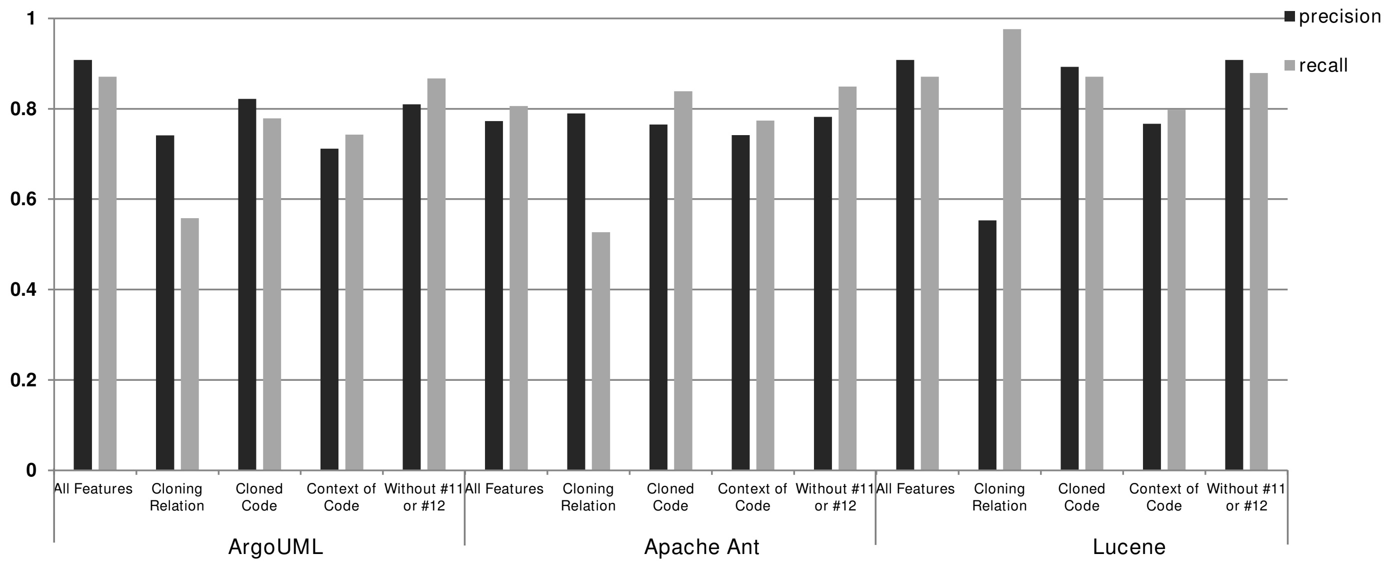


Fig. 3: Performance of using subsets for classification training for non-refactoring group

recall. To test whether classifiers trained in this work suffer from the problem of model overfitting, we have conducted cross-project validation for the classification models trained on each target system. Therefore, each classification model is tested on data that is previous unknown to it during model building. The equally good performance in cross-project validation indicates that our model of clone refactoring recommendation can be generalized to apply on other systems.

C. Unrefactored Clones

In this paper, classification models are trained based on the historical refactoring of clones. We believe that whether a clone class is refactored in practice reflects the benefits, costs, and the risks a potential refactoring entails. However the clone refactoring history can also be influenced by other factors, and the actual decision might not represent the best practices of clone maintenance. It is thus possible that some of the unrefactored clones should have been refactored. In our within-project testing of ArguUML, we have found one unrefactored clone instance that is $I_{n \to r}$ (incorrectly labeled as "refactoring"). We have a surprising finding when manually examining this clone instance: Clone evolution indicates that this clone is never changed (see Section III-A); however, the comments precedes the cloned code snippet indicates otherwise:

```
// Refactoring: static to denote that it doesn't
// use any class members.
// TODO:
// Idea to move this to MMUtil together with the
// same function from
// org/argouml/uml/cognitive/critics/WizOperName.java
// org/argouml/uml/generator/ParserDisplay.java
```

The comments evidently suggest that developers are aware of the clones spanning the two files, and one developer has explicitly expressed the intention to extract a method into a utility class. However, subsequent version evolution indicates that this proposed refactoring never happened, therefore, we do not consider these clones as "refactored" in our training data. This may explain some instances that are incorrectly labeled as "refactoring" ($I_{n \rightarrow r}$ instances). The possible remedy in this case is manual labeling of each clone, based on the analysis of benefits, cost, and risk that a potential clone refactoring entails.

D. Incorporating With Other Research

By correctly recommending clones for refactoring, our approach can be incorporated with prior research. Higo et al. [17] proposed a metric-based approach to suggest what refactoring methods can be applied to a clone. Our work, which recommends *which* clones for refactoring, allows the approach by Higo et al. to be applied on only a small proportion of clone instances, resulting in better usability for developers.

VI. Threats to Validity

Construct. In this paper, we have identified clone refactoring cases by a combination of metrics that indicates clone refactoring. The preservation of functionality of clone evolution is determined manually. Our approach is justified since it can capture more clone refactoring instances than every other approach to identify refactoring instances.

External. We investigate only open source projects that are written in Java. Further investigation is required to validate our approach on software projects that are developed with other settings, such as programming languages, development styles, or application domains.

Clone Refactoring History. In this study, we use only the history of clone refactoring to train and evaluate classifiers. However, refactoring history may not represent the best practice of clone refactoring. Developers may be incentivized to work on clones that are *convenient* for refactoring (i.e., are "low-hanging fruit"). Likewise, developers may be unable to determine what is best for long-term software quality, especially when software systems are undergoing rapid evolution. Consequently, our classification model may reflect some decision elements that are not necessarily best for long-term software quality. To account for this, we can ask experts to label each clone according to an explicitly defined model for benefit, cost and risk of clone refactoring.

To free experts from manually labeling thousands of clones, we can build our classifier using a semi-supervised learning approach [4], where labeled data and input from expert can be used in conjuncture to iteratively update our classification model. However, incorporating experts' input into the training process also introduces new sources of bias.

VII. Related Work

A. Managing Software Clones

Researchers have proposed and implemented various techniques that allow for visualization or tracking of software clones. Duala-Ekoko and Robillard [10] design a tool "*CloneTracker*" to track specific clones over versions within the Eclipse platform. Duala-Ekoko and Robillard define an abstraction for surroundings of clones so that "CloneTracker" can tolerate location changes of clones over versions. In their work, the support for simultaneous editing of a clone class is also provided based on comparing Levenshtein distance between each line of source code.

Hou et al. [19] also design and implement a tool which captures editing activities of developers. After capturing the creation of clones (the copy-and-paste activities in the Eclipse environment), their tool tracks the evolution of clones by collecting upcoming editing activities of clones. Nguyen et al. [33] develop an AST-based tool that supports change consistency validation of clones, clone merging, clone synchronization for developers. Lin et al. [30] also design an Eclipse plugin, which automatically computes and highlights syntactical differences among clone fragments to help clone refactoring for developers.

Balazinska et al. [1] design an AST-based tool that can transform clones into one of two object-oriented design patterns: Strategy or Template Method. Higo et al. [17] identify several metrics which indicate refactoring opportunities, which are used to automatically recommend refactoring methods for each clone class.

More discussions of existing work and possible research directions of clone management can be found in a survey by Koschke [27], and another survey by Roy et al. [39].

B. Feature-based Analyses of Software Clones

Researchers have previously applied feature-based classifiers on software clones. Steidl and Göde [43] use a decision tree-based classifier to predict bugs in clones in four large proprietary systems. In their work, they select thirteen features from two dimensions: source code of cloned code and inconsistency among clone siblings.

Xiaoyin Wang et al. [46] construct a Bayesian Network to predict evolution consistency of a potential clone when developers are trying to introduce new clones. Their training data is based on 21 features from three dimensions: history features, source code features of cloned code, and features of destination of the new clone from two large proprietary systems.

Zibran et al. [48] propose a model to estimate the efforts of clone refactoring. Mandal et al. [31] use history of clone changes to infer co-change association among clone classes.

C. Evolution of Software Clones

Choi et al. [5] conducted a case study of clone refactoring history from three open source Java systems, and identified several refactoring patterns in clone evolution. Yoshida et al. [47] surveyed existing research in code refactoring, and suggested research directions toward improved solution to clone refactoring.

To determine clone genealogy over versions, Kim et al. [25] designed a tool which relies on source file locations to identify clones over versions. Krinke [28] studied the consistent and inconsistent changes to clones on five open source systems written in Java or C++, observing that half of clone classes had inconsistent changes. Krinke [29] also investigated the stability of cloned code compared to non-cloned code. Saha et al. [41] conducted an empirical study on the evolution of Type-3 clones and discovered a higher proportion of inconsistent changes for Type-3 clones.

Through a case study of several large open source systems, Kapser et al. [23], [24] summarize patterns of clones according to the intention of code duplication. They provide empirical evidence indicating that cloning can be a principled engineering tool, thus providing early evidence that clones are not always harmful. Kapser et al. also categorize clones into several patterns. It is worth noticing that certain patterns are particularly suitable for refactoring, while other patterns are not. For example, they observe that for the pattern "Verbatim Snippets", it contains "*repetitive fragments of logics must be reused*", and thus conclude that clones belonging to this pattern "*should be factored out as helper functions*".

Thummalapenta et al. [44] performed an empirical study on clone evolution, especially on change consistency or late propagation in cloned code. Göde [12] proposed and implemented a novel approach to detect clone removal cases, and investigated nine projects of change consistency in clone evolution.

Clone removal has been a focus of some recent empirical studies. Göde [13] conducted a case study on clone removal, where the clone removal instances are inferred by analyzing the change of size of clone fragments. The distribution of clone removal methods is also reported in this paper. In this study, Göde also investigated the feasibility of using commit logs to identify clone removal instances. Göde and Koschke [15] have also conducted a study on clone evolution, and have observed that only a small number of inconsistent changes to clones is unintentional. Bazrafshan et al. [2] further investigated accidental clone removals, reporting the types of clone refactoring in eleven open source systems in Java or C/C++.

VIII. Conclusion

Cloning is often used as a convenient design shortcut to reuse an existing solution by replicating and then specializing code fragments within a software system. However, cloning may cause long-term software maintenance issues or even introduce bugs. Clone refactoring, the primary method to eliminate clones, remains difficult as there are no existing approaches to recommend clones for refactoring.

In this paper, we construct a machine learning classifier to recommend clones for refactoring. Our classifier is constructed by learning from clone refactoring history. Fifteen features are extracted from clones so that our classifier can reflect the benefit, cost, and risk of clone refactoring. To collect training data for classifier learning, we use a metrics-based approach which identifies 323 clone refactoring instances from three open source projects. We also select an equal number of clone instances that are without refactoring to the training data.

For 646 total clone instances in our training data, within-project testing generates precision from 77.3 % to 87.9% in 10-fold cross validation. Cross-project testing also yields results with good performance: the lowest precision is 73.2%, and the highest is 88.5%. To understand the effectiveness of specific types of features, we also report the performance of classifier training by only subsets of features. Results suggest that features that are related with cloned code snippets can already yield good results, although further performance improvement is gained by adding features of clone context and cloning relations. The overall performance of our classifier suggests that by learning from features of clones, we can recommend clones for refactoring with a high precision and recall according to clone refactoring history. By recommending which clones are appropriate to refactor, our work allows for better resource allocation for clone refactoring, leading to improved clone management.

Acknowledgment

We thank Rainer Koschke and his colleagues for providing iClones, which is used to perform clone detection and clone genealogy analysis in our work. We also thank Haroon Malik and Cheng Zhang for their comments and discussions. Thanks to the anonymous reviewers for their valuable feedback on the paper.

References

[1] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pages 98–107, 2000.

[2] S. Bazrafshan and R. Koschke. An empirical study of clone removals. In *29th IEEE International Conference on Software Maintenance (ICSM)*, pages 50–59, Sept 2013.

[3] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, Sept 2007.

[4] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100. ACM, 1998.

[5] E. Choi, N. Yoshida, and K. Inoue. What kind of and how clones are refactored?: A case study of three oss projects. In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 1–7, New York, NY, USA, 2012. ACM.

[6] J. R. Cordy. Comprehending reality " practical barriers to industrial adoption of software maintenance automation. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, IWPC '03, pages 196–, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Y. Dang, S. Ge, R. Huang, and D. Zhang. Code clone detection experience at Microsoft. In *IWSC*, pages 63–64, 2011.

[8] Y. Dang, D. Zhang, S. Ge, C. Chu, Y. Qiu, and T. Xie. Xiao: Tuning code clones at hands of engineers in practice. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 369–378, New York, NY, USA, 2012. ACM.

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 166–177, New York, NY, USA, 2000. ACM.

[10] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[11] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[12] N. Göde. Evolution of type-1 clones. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 77–86, Sept 2009.

[13] N. Göde. Clone removal: Fact or fiction? In *Proceedings of the 4th International Workshop on Software Clones*, IWSC '10, pages 33–40, New York, NY, USA, 2010. ACM.

[14] N. Göde and R. Koschke. Incremental clone detection. In *13th European Conference on Software Maintenance and Reengineering*, pages 219–228, March 2009.

[15] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *33rd International Conference on Software Engineering*, pages 311–320, May 2011.

[16] H. He and E. Garcia. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21(9):1263–1284, Sept 2009.

[17] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(6):435–461, 2008.

[18] A. Ho. Visual studio ultimate 2012: Finding and managing cloned code. http://channel9.msdn.com/Series/Visual-Studio-2012-Premium-and-Ultimate-Overview/Visual-Studio-Ultimate-2012-Finding-and-managing-cloned-code, Microsoft Corporation, accessed in April 2014.

[19] D. Hou, P. Jablonski, and F. Jacob. Cnp: Towards an environment for the proactive management of copy-and-paste programming. In *IEEE 17th International Conference on Program Comprehension*, pages 238–242, May 2009.

[20] J. Jang, A. Agrawal, and D. Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *IEEE Symposium on Security and Privacy (SP)*, pages 48–62. IEEE, 2012.

[21] G. H. John, R. Kohavi, K. Pfleger, et al. Irrelevant features and the subset selection problem. In *ICML*, volume 94, pages 121–129, 1994.

[22] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, July 2002.

[23] C. Kapser and M. Godfrey. "cloning considered harmful" considered harmful. In *13th Working Conference on Reverse Engineering*, pages 19–28, Oct 2006.

[24] C. Kapser and M. Godfrey. "cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Softw. Engg.*, 13(6):645–692, Dec. 2008.

[25] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.

[26] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 50:1–50:11, New York, NY, USA, 2012. ACM.

[27] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software, Dagstuhl Seminar Proceedings 06301*, 2006.

[28] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 170–178, Washington, DC, USA, 2007. IEEE Computer Society.

[29] J. Krinke. Is cloned code more stable than non-cloned code? In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 57–66, Sept 2008.

[30] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao. Detecting and summarizing differences across multiple instances of code clones. In *36rd International Conference on Software Engineering (ICSE)*, June 2014.

[31] M. Mandal, C. Roy, and K. Schneider. Automatic ranking of clones for refactoring through mining association rules. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week*, pages 114–123, Feb 2014.

[32] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin. Gathering refactoring data: A comparison of four methods. In *Proceedings of the 2nd Workshop on Refactoring Tools*, WRT '08, pages 7:1–7:5, New York, NY, USA, 2008. ACM.

[33] T. T. Nguyen, H. A. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone-aware configuration management. In *24th IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134, Nov 2009.

[34] C. Parnin and C. Görg. Improving change descriptions with change contexts. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pages 51–60, New York, NY, USA, 2008. ACM.

[35] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, Sept 2010.

[36] J. R. Quinlan. *C4.5: programs for machine learning*, volume 1. Morgan Kaufmann, 1993.

[37] C. Roy and J. Cordy. An empirical study of function clones in open source software. In *15th Working Conference on Reverse Engineering*, pages 81–90, Oct 2008.

[38] C. Roy and J. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *The 16th IEEE International Conference on Program Comprehension*, pages 172–181, June 2008.

[39] C. Roy, M. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future (keynote paper). In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week*, pages 18–33, Feb 2014.

[40] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *School of Computing Technical Report 2007-541, Queen's University*, 115, 2007.

[41] R. Saha, C. Roy, K. Schneider, and D. Perry. Understanding the evolution of type-3 clones: An exploratory study. In *10th IEEE Working Conference on Mining Software Repositories*, pages 139–148, May 2013.

[42] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue. Finding file clones in freebsd ports collection. In *7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 102–105, May 2010.

[43] D. Steidl and N. Göde. Feature-based detection of bugs in clones. In *IWSC '13*, pages 76–82, 2013.

[44] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.

[45] W. Wang and M. Godfrey. Investigating intentional clone refactoring. In *Proceedings of the 8th International Workshop on Software Clones*, IWSC '14, 2014.

[46] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can i clone this piece of code here? In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 170–179. ACM, 2012.

[47] N. Yoshida, E. Choi, and K. Inoue. Active support for clone refactoring: A perspective. In *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools*, WRT '13, pages 13–16, New York, NY, USA, 2013. ACM.

[48] M. F. Zibran and C. K. Roy. A constraint programming approach to conflict-aware optimal scheduling of prioritized code clone refactoring. In *Proceedings of the 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, SCAM '11, pages 105–114, Washington, DC, USA, 2011. IEEE Computer Society.