Tool Support for Managing Clone Refactorings to Facilitate Code Review in Evolving Software

Zhiyuan Chen[1], Maneesh Mohanavilasam[2], Young-Woo Kwon[2], Myoungkyu Song[1]
[1]University of Nebraska, Omaha, NE, USA [2]Utah State University, Logan, UT, USA
zhiyuanchen@unomaha.edu maneesh.m@usu.edu young.kwon@usu.edu myoungkyu@unomaha.edu

***Abstract*—Developers often perform copy-and-paste activities. This practice causes the similar code fragment (aka code clones) to be scattered throughout a code base. Refactoring for clone removal is beneficial, preventing clones from having negative effects on software quality, such as hidden bug propagation and unintentional inconsistent changes. However, recent research has provided evidence that factoring out clones is not always to reduce the risk of introducing defects, and it is often difficult or impossible to remove clones using standard refactoring techniques. To investigate which or how clones can be refactored, developers typically spend a significant amount of their time managing individual clone instances or clone groups scattered across a large code base. To address the problem, this paper presents a technique for managing clone refactorings, Pattern-based clone Refactoring Inspection (PRI), using refactoring pattern templates. By matching the refactoring pattern templates against a code base, it *summarizes refactoring changes of clones*, and *detects the clone instances not consistently factored out* as potential anomalies. PRI also provides novel visualization user interfaces specifically designed for inspecting clone refactorings. In the evaluation, PRI analyzes clone instances in six open source projects. It identifies clone refactorings with 94.1% accuracy and detects inconsistent refactorings with 98.4% accuracy. Our results show that PRI should help developers effectively inspect evolving clones and correctly apply refactorings to clone groups.**

I. Introduction

Code changes are usually repetitive. Recent research [20], [34] has pointed out that over 30% of the total amount of code is repetitive regarding various application domains, such as operating systems, web server programs and development environments. Changing the code similarly is an easy way to achieve a design goal in adding new features and fixing numerous bugs, mostly because of the copy-paste(-and-adapt) programming practice, the framework-based development, and the reuse of the same design patterns or libraries, thus creating *code clones*.

While several studies [23], [24], [13] find positive aspects of code clones, in other studies [18], [2], [32], cloning is considered harmful to software quality, leading to much effort on detection and removal of code clones. For example, anomalous changes could repeat either by a developer's own error or by other developers' fault unknowingly [2]. Code clones also require that changes to one section of code clones are to be propagated to multiple locations consistently, incurring additional maintenance costs to synchronize cloned regions [18], [32].

Despite high performance in detecting code clones [40], understanding clone groups (i.e., sets consisting of two or more clone instances) remains challenging. To improve understandability and maintainability, clone management tools have been developed. These techniques represent differences across multiple clone instances [28], track evolving code clones in a repository on a Version Control System (VCS) [7], and assist changes of clones and their contexts [16].

In management of code clones, developers often conduct *clone refactoring*, combining regions of code that are very similar and moving them into a function without altering the existing functionality. To help developers conduct refactorings for clone removal, Integrated Development Environments (IDEs), such as Eclipse, provide automated refactoring features. Unfortunately, not all developers make consistent use of these features, since refactorings are not always feasible by standard refactoring engines in IDEs [20]. Professional developers also tend not to use automated refactoring despite their awareness of the refactoring features of IDEs [45]. These manual refactorings often lead to error-proneness. Recent studies [19], [36] report that the ratio of refactorings affects increasing numbers of bugs, and Park et al. [36] find that regarding an omission error type such as incomplete refactorings, it takes much longer to be resolved than other types of defects. As a result, it is difficult and time-consuming for code reviewers to inspect individual or a group of clone instances and answer questions such as "How should these clones be refactored completely or correctly?" and "Are there any clones that should be removed along with these clones?".

This paper proposes a technique for inspecting refactorings of evolving clones and detecting incomplete refactorings: **P**attern-based clone **R**efactoring **I**nspection (PRI). To examine which or how clones are refactored and to determine whether clones are refactorable, we implemented refactoring pattern templates, which are automatic Abstract Syntax Tree (AST) matching programs based on standard refactoring types from Fowler's catalog [10]. We target five refactoring types: Extract Method, Pull Up Method, Move Method, Extract Super Class, and Move Type To New File, and two composite refactorings: Extract & Move Method and Extract & Pull Up Method. PRI is used as follows: (1) When a developer begins a code review on a program, PRI takes as input the results from a clone detector [17], which runs on the source code of her revision. (2) It then automatically identifies refactored regions through refactoring pattern rules in the subsequent revisions, incrementally tracking the clones in a repository. (3) Finally, PRI *summarizes* refactoring changes across revisions and *detects*

0730-3157/17 $31.00 © 2017 IEEE
DOI 10.1109/COMPSAC.2017.242

clone instances which developers may overlook to refactor or update inconsistently and result in incomplete refactorings as potential refactoring anomalies,[1] leading to difficulties in managing the clone group. It also *visualizes* clone refactorings and anomalies capable of tracking their change histories.

PRI summarizes groups of clones, including their refactored revisions and corresponding change information (e.g., refactoring type, changed location and program restructuring description), and classifies unrefactored code clones (e.g., refactorable clones and a recommended refactoring type or unfactorable clones and a discovered reason of not being refactorable). PRI is an Eclipse plug-in that builds on top of our AST-based code search technique.[2]

To demonstrate the benefits of our approach, we conducted two case studies. First, we evaluated the accuracy of PRI's clone refactoring summarization. We manually constructed the ground truth of refactoring changes to clones on a sampled data set from six open source projects. The comparison between the PRI's results against this ground truth found that it demonstrates precision of 98.9% and recall of 92%.

In our second case study, we applied PRI to a data set with incomplete refactorings of clones performed by real developers. PRI can detect such clone refactoring anomalies with 99.4% precision and 97.8% recall by automatically tracking clones and the corresponding refactorings across revisions.

Overall, this paper makes the following contributions:

- A new approach for inspecting clones for refactoring. PRI provides a novel integration of program differencing and AST-based code pattern search to track the changes to clones based on well-known clone removal refactorings [33], [45]. It also detects unrefactored clones, extracts clone differences, and classifies these clones to provide a related reason of not being refactorable.
- We implemented our technique in an open-source plug-in for the Eclipse IDE. This plug-in, also called PRI, supports code reviewers with a summarization of clone refactorings and detection of incomplete refactorings of clone groups as potential anomalies, while displaying the refactoring edit history on a visualization view.
- We evaluated PRI by conducting two case studies with six open source projects. In these studies, we measured PRI's summarization capability regarding clone refactorings and the detection capability regarding incomplete refactorings of clone groups.

II. Related Work

Existing code review tools, such as Codeflow, Collaborator, Gerrit, and Phabricator[3] are usually used in practice but require exploring each line file by file, even though cross-file changes are made with code clones. Unlike PRI, developers are left to manually find other locations of clone siblings that are refactored similarly.

[1]In this paper, we interchangeably use the terms bug, fault, anomaly and refactoring problem.

[2]http://faculty.ist.unomaha.edu/msong/pri/

[3]https://goo.gl/xaE4Pe, https://goo.gl/oBz3ea, http://phabricator.org, and http://code.google.com/p/gerrit/

Several approaches detect inconsistencies in clones. CP-Miner [27], SecureSync [37] and Jiang et al.'s technique [18] reveal clone-related bugs by finding recurring vulnerable code. CBCD reveals bug propagation in clones by finding similar ASTs in a program dependence graph [26]. SPA analyzes discrepancies in changes and detects inconsistent updates in clones [39].

Our approach differs from these inconsistency detection techniques in two ways. First, PRI automatically accesses to VCS and incrementally identifies inconsistent changes in clone histories, unlike analysis of one or two versions. Due to these differences, in our case studies, we could not directly compare with existing clone-based code search techniques since these tools are not designed for inspecting clone evolution and its refactoring edits. Second, in contrast to PRI's refactoring classification, the clones found by these tools require manual inspection to determine if these clones can be removed using standard refactoring techniques [10].

Göde and Koshke [14] present an incremental clone detection algorithm to study clone evolution and find that most clones remain unchanged during their lifetime, and clones are mostly changed inconsistently. Krinke's [22], [23] study investigates the changes to clones in five open source systems and find that some clone groups are changed consistently. Saha et al. [41] investigate the evolution of clones, and their results show that clone type is more likely to change inconsistently, when clones form gaps among clone fragments. Aversano et al. [1] study how clones are evolved and find that developers almost propagate the change consistently. Bettenburg et al. [5] investigate the effect of inconsistent changes to code clones and observe that 1–3% of inconsistent changes to clones introduce defects, reporting that most of clones are consistently maintained. Although these approaches map clones between revisions of a program, they do not provide summarization of changes to clones for investigating clone refactorings.

These studies show that removing code clones is not always necessary nor beneficial. PRI helps to summarize clone evolution and refactorings, which developers can investigate during peer code reviews.

Toomim et al. [43], Duala-Ekoko and Robillard [7], Hou et al. [16] and Nguyen et al. [35] manage clones in evolving software by tracking the changes in the clones as code evolves. Lin et al. [28] also design a plug-in built on Eclipse IDE that computes differences among clones and highlights the syntactic differences across multiple clone instances. Unlike the above approaches, PRI leverages the clone region information to help code reviewers detect incomplete refactorings of clone groups which are omission-prone, supporting *clone-aware refactoring recommendations*.

Tsantalis et al. [44] use a program slicing technique to capture code modifying an object state and design rules to identify refactoring candidates from slices. Bavota et al.'s [4] approach identifies classes to extract by using similarity and dependence between methods in a class. While these tools identify refactoring opportunities for clones, they do not support developers with *real* refactoring examples. Our approach

provides concrete information of clone differences showing real refactoring examples of other siblings in the group.

REFFINDER [38] could be used to search for refactorings. However, we find that it includes false negative refactoring cases such as clones that were factored out into nested method calls. In contrast to analysis of only VCS data in REFFINDER, PRI analyzes both clone groups in clone database and source code in VCS to capture more precise data for identifying clone refactorings.

BeneFactor [12] and WitchDoctor [9] use refactoring patterns to help developers complete refactorings that were started manually. PRI uses refactoring patterns, but these patterns are automatically matched with clones across revisions.

Kim et al. [20] study the evolution of clones and provide a manual classification for evolving code clones. Unlike their approach, we automatically classify clones if they are not easily refactorable using standard refactoring techniques [10].

III. MOTIVATING EXAMPLE

This section describes PRI with a real example drawn from the JEdit (http://jedit.org) project, which is an open source project—a text editor written in Java—with 120K lines of source code over 580 Java source files.

Suppose Alice changes JEdit's source code after she has encountered duplicated regions. She manually applies Extract Method to two cloned regions in methods `processKeyEvent` and `processKeyEventV2`[4], respectively. She validates manual refactorings [45] using existing test cases; however, she misses refactoring one clone instance of the group in a different method `processMouseEvent` in Figure 1(c). For another clone instance in `processActionEvent` in Figure 1(d), she has difficulty conducting the same refactoring due to the type variation of variable `changeEventAction`, which is different from corresponding variables of other clone siblings in the same group.

To confirm that there is no location Alice missed to refactor during peer code review, Barry needs to investigate line level differences file by file. When Barry finds suspicious locations, he might want to inspect differences between Alice's and subsequent revisions. Simply comparing with the newest revision can require Barry to decompose countless irrelevant changes [3], [42]. Since understanding such composite changes require non-trivial efforts [42], sub-changes that are aligned with Alice's refactoring changes must be investigated by manually comparing Alice's changes with other changes committed in subsequent revisions.

The following shows how Barry may use PRI to inspect Alice's changes. First, he checks out revision r_i that she commits changes, and then he runs PRI, a plug-in built atop Eclipse IDE. Based on r_i, PRI tracks r_i, r_{i+1}, .., r_{i+n}, and summarizes clone refactorings performed by Alice (Figure 1(a)) and applied by other developers reusing the existing code by altering copy-pasted code (Figure 1(b)). The results include refactored revisions, types (e.g., Extract Method), locations (e.g., package, class, method, and line number) and restructuring descriptions (e.g., "Method `m1` is refactored and

[4]Method `processKeyEventV2` is created to support the different version.

```
1  public void processKeyEvent(KeyEvent evt, int from) {
2     Event focusKeyTyped, focusKeyPressed;
3     switch(evt.getID()) {
4     case Event.KEY_TYPED:
5   - if(inputHandler.isActive() && from != VK_CANCEL) { ..
6   -   focusKeyTyped = evt.getEvent();
7   -   ..}
8   +  focusKeyTyped = processEvent(from, focusKeyTyped);
9   ..
10    case Event.KEY_PRESSED:
11  - if(inputHandler.isActive() && from != VK_CANCEL) { ..
12  -   focusKeyPressed = evt.getEvent();
13  -   ..}
14  +  focusPressed = processEvent(from, focusKeyPressed);
15  ..} ..}
16
17  + Event processEvent(int from, Event event) {
18  +  if(inputHandler.isActive() && from != VK_CANCEL) { ..
19  +    focusKeyTyped = evt.getEvent();
20  +    ..}
21  +  return event;
22  + }
```

(a) A clone refactoring in revisions v20060919-7074 and v20060919-7075 in JEdit.

```
1  public void processKeyEventV2(KeyEvent evt, int from, int enhancedVer) {
2     Event focusKeyTypedEvt, focusKeyPressedEvt;
3     switch(evt.getID()) {
4     case Event.KEY_TYPED:
5   - if(inputHandler.isActive() && from != VK_CANCEL) { ..
6   -   focusKeyTypedEvt = evt.getEvent();
7   -   ..}
8   +  focusKeyTypedEvt = processEvent(from, focusKeyTypedEvt);
9   ..
10    case Event.KEY_PRESSED:
11  - if(inputHandler.isActive() && from != VK_CANCEL) { ..
12  -   focusKeyPressedEvt = evt.getEvent();
13  -   ..}
14  +  focusKeyPressedEvt = processEvent(from, focusKeyPressedEvt);
15  ..} ..}
```

(b) A refactoring to clones of the same group in a later revision by other developer.

```
1  void processMouseEvent(MouseEvent evt, int src) {
2     Event focusMousePressed;
3     switch(evt.getID()) {
4     case Event.MOUSE_PRESSED:
5     if(inputHandler.isActive() && src != VK_CANCEL) { ..
6        focusMousePressed = evt.getEvent();
7     ..}
8  ..} ..}
```

(c) A clone instance that Alice misses to apply a same refactoring as (a) and (b).

```
1  void processActionEvent(ActionEvent evt, int where) {
2     Action changeEventAction;
3     switch(evt.getID()) {
4     case Event.CTRL_MASK:
5     if(inputHandler.isActive() && where != VK_CANCEL) { ..
6        changeEventAction = evt.getEvent(); // The type of changeEventAction differs
                 from aligned variables in other siblings in the clone group.
7     ..}
8  ..} ..}
```

(d) A clone instance in the clone group, which is not refactorable unlike (a) ∼ (c).

Fig. 1: A simplified example: A clone group is refactored inconsistently. Cloned regions are highlighted, deleted code is marked with '-' and added code marked with '+'.

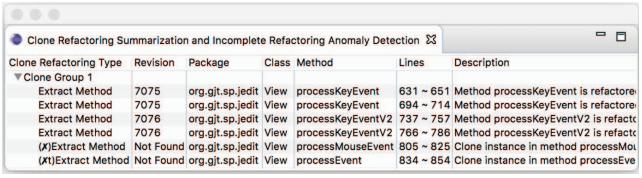


Fig. 2: A viewer in PRI that shows clone refactoring summarization and refactoring anomaly detection regarding Figure 1.

cloned code fragments are replaced with a call to an extracted method `m2`"). PRI also detects unrefactored clones in a clone group (Figure 1(c)), classifying if a clone instance is locally refactorable by standard refactorings [10] (Figure 1(d)).

Figure 2 shows a snapshot of a PRI's viewer, which shows a clone group that comprises six clone instances searched in each row of a tree viewer. The first two rows show the clone instances factored out by Extract Method in revision 7075, and the next two rows show the clone instances refactored in the

same way in the next revision 7076 (Section IV-A). PRI marks the fifth clone instance with symbol ✗, meaning that it detects an omission error (Figure 1(c)). It marks the last clone instance with symbol ✗t, meaning that it discovers Alice's difficulty in applying Extract Method in the same way due to the type variation (Figure 1(d)) (Section IV-B).[5] The viewer in Figure 2 is synchronized with a visualization view (Section IV-C) designed for inspecting clone refactorings regarding structural and dependence relationships between clone instances and related contexts.

The aforementioned incomplete refactorings do not produce compilation errors, passing all existing test cases. Reviewers are likely to overlook these locations. In fact, it is not always possible to factor out all clones but independent evolution might be required in some clone instances; however, revealing these locations and classifying them whether to easily be removed using standard refactoring techniques can be worthwhile during a code review.

IV. PRI:PATTERN-BASED CLONE REFACTORING INSPECTION

This section presents PRI consisting of the following three phases. Phase I summarizes clone refactorings using templates, which check the structural constraints before and after applying a refactoring to a program and encode ordering relationship between refactoring types. Phase II detects clone groups incompletely refactored, classifying the reasons. Phase III provides code visualization to represent historical refactoring edits that Phases I finds and refactoring opportunities that Phases II detects. Figure 3 outlines the workflow of PRI.

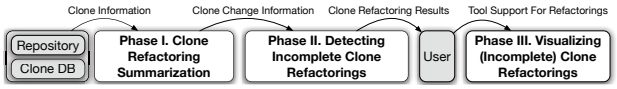


Fig. 3: Overview of PRI's workflow.

A. Phase I: Change Summarization for Clone Refactorings

Converting Clone to AST Model. Our approach parses clone groups reported by *Deckard* a clone detector [17].[6] PRI finds the set of AST nodes that contains reported clones:

$$\{n \mid offset(clone) \geq offset(statements) \wedge length(clone) \leq length(statements) \wedge n \in ast(statements)\} \quad (1)$$

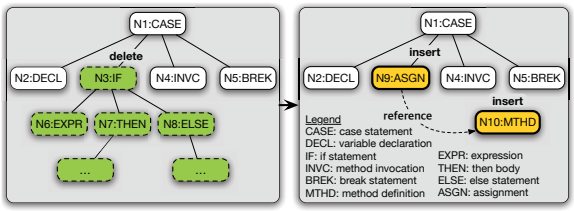
where *offset(clone)* is the starting position of clone instance *clone* from the beginning of the source file, and *length(clone)* is the length of the clone instance. The function *ast(statements)* finds an AST at the method level, and analyzes the AST to find inner-most syntactic statements (i.e., least common ancestor) that may contain incomplete syntactic regions of cloned code fragments. PRI improves the performance of search by caching ASTs of syntactic clones after computing Equation 1. The tool for parsing Java source code and generating the corresponding ASTs is provided with the Eclipse JDT framework.[7]

Tracking Clone Histories. PRI accesses each subsequent revision $r_j \in R = \{r_{i+1}, .., r_n\}$, and identifies ASTs that are related to clone instances in an original revision r_i.

PRI presents a *clone refactoring aware approach* to checking clone change synchronization across revisions. It is integrated with software configuration management (SCM) tools using an SCM library[8] to analyze refactorings of individual clones and groups, helping developers focus their attention on any revision.

Returning to the motivating example, CI_1 in `processKeyEvent` is a clone of a fragment CI_2 in `processKeyEventV2` in the clone group. At revision r_1, changes of Extract Method to CI_1 are checked in, and changes of the same refactoring to CI_2 are committed to the next revision r_2. To track the evolved clone group, PRI automatically accesses the consecutive revisions (n $\geq$ 2, where n is configurable) to search for the clone siblings (CI_1 and CI_2) by comparing two revisions.

Extracting Changes between Two Versions of ASTs. PRI leverages ChangeDistiller [8] to compute AST edits to code clones. We chose ChangeDistiller since it represents concise edit operations between pairs of ASTs by identifying change types such as adding or deleting a method, inserting or removing a statement, or changing a method signature. It also provides fine-grained expression with a single statement edit. PRI uses ChangeDistiller to compute differences between the clone ASTs in revision r_i and the evolved clone ASTs in revision r_j.



(a) Before refactoring a clone instance (b) After refactoring a clone instance

Fig. 4: Extracting the edit operations from clone changes.

Continuing with our motivating example, PRI parses the clone region before changes as shown in Figure 4(a), and parses the corresponding region after changes as shown in Figure 4(b). It then uses ChangeDistiller to compute the differences between two sets of ASTs. In Figure 1(a), ChangeDistiller reports the deletion operations of the statements including `if`, `then` and `else`, and the insertion operations of an assignment statement including a call to a new method `processEvent`.

When checking changes before and after clone refactorings, it is important to determine if references (e.g., method call and field access) are preserved across clone instances. Therefore, we create a new reference binding checker, which is not provided by ChangeDistiller, to assess reference consistency. For

[5]Java generic types could be applicable for a refactoring, which is not included in standard refactorings [10].

[6]The default settings with 30 minT (minimal number of tokens required for clones), 2 stride (size of the sliding window), and 0.95 *Similarity* are used.

[7]http://www.eclipse.org/jdt/

[8]http://www.svnkit.com/

example, Figure 4(b) shows a method invocation dependency between a caller in node N_9 and a callee in N_{10}. We check if this reference association is preserved in other regions after changing clone instances by using bindings to a method.

Matching Clone Refactoring Pattern Templates. Refactoring pattern templates are AST-based implementations that consist of a pair of *pre- and post-edit matchers* such as $\mathcal{M}_{pre}$ and $\mathcal{M}_{post}$. $\mathcal{M}_{pre}$ is an implementation for matching patterns before clone refactoring application. $\mathcal{M}_{post}$ interacts with repositories and traverses ASTs of the source code in which the clones and their dependent contexts are modified. It extracts both a *match* between the nodes of the compared AST subtrees before and after a refactoring application and an *edit script* of tree edit operations transforming an original into a changed tree.

After matching such *clone refactoring patterns* comprising a set of constraint descriptions where a refactoring can be performed, PRI identifies concrete clone refactoring changes (e.g., refactored revisions, refactoring types, locations, and restructuring descriptions). The change pattern descriptions are designed by using declarative rule-based constraints [38].

Table I shows clone refactoring templates that our approach can identify based on pre- and post-edit matchers. We leverage the rules of refactoring types in Fowler's catalog [10]. A composite refactoring comprises a set of low-level refactorings. For example, template 6 describes that Extract Method is applied to a clone group, and the new method is moved to another class.

ID		Type	Template
1	EM	Single Refactoring	Extract Method Refactoring
2	MM		Move Method Refactoring
3	PM		Pull Up Method Refactoring
4	ES		Extract Superclass Refactoring
5	MN		Move Type to New File Refactoring
6	EM+MM	Composite Refactorings	Extract and Move Method Refactoring
7	EM+PM		Extract and Pull Up Method Refactoring
8	PM+EM		Pull Up and Extract Method Refactoring

TABLE I: Clone refactoring templates that PRI supports by tracking the evolution of clones focusing on their removal refactorings.

Algorithm 1: Identifying Clone Refactoring Evolution.

Input : *REVs* – a scope of revisions for inspection, *OCD* – the output of a clone detector, and *PRG* – a program to be inspected.
Output: *RES* – a set of clone groups whose clone instances are classified as refactored, refactorable, or unfactorable.

```
1 Algorithm main
2     CGs ← M_pre.match (OCD, PRG);
3     foreach Revision r_i ∈ REVs do
4         foreach CloneGroup G_clone ∈ CGs do
5             RES ← M_post.match (G_clone, r_i);
6         end
7     end
```

Algorithm 1 illustrates our approach following this clone refactoring pattern identification. Our approach first takes a revision scope (REVs) for tracking clone refactoring histories and the output of a clone detector (OCD) as input. To match pre-edit patterns with OCD, it traverses the ASTs of OCD and extracts program elements (e.g., packages, classes, methods, and fields) and structural dependencies (e.g., containment, subtyping, overriding, and method calls). $M_{pre}.match$, a syntactic sugar, uses these predicates to determine clone structural patterns, such as whether they exist in the same structural location, or whether they are implemented in classes with the common superclass (Line 2). It then iterates two tasks: tracking clone groups across revisions (Line 3) and inspecting each clone group (Line 4). This iteration stops when all changed files in input revisions are inspected with all clone groups.[9] Algorithm 1 returns the following results: (1) clone groups where all clone instances are refactored completely, (2) clone groups where no clone instance is refactored, and (3) clone groups where some of the clone instances are refactored. $M_{post}.match$ is a syntactic sugar for the invocation of any template to match with changes to clones (Line 5).

Continuing with our example, PRI matches patterns with changes to clones (Figures 1(a) and (b)) in each revision using templates. During this inspection, Algorithm 2 finds the pattern of Extract Method by performing rules in lines 3-4.

Algorithm 2: Matching the Extract Method Refactoring Pattern.

Input : *G* – a clone group, *R* – a revision, and *Sim* – a similarity threshold.
Output: *RES* – clone instances identified as refactored or not.

```
1  Template extractMethodRefPattern
2      foreach CloneInstance ci ∈ G do
3          if deleteClone (ci, m_i, t_i, R) ∧ addMethod (m_j, t_i, R) ∧
4          addCall (m_i, m_j, R) ∧ similarBody (m_i, m_j, Sim) then
5              RES ← summarize (ci); // clone refactoring
6          end
7          else
8              RES ← detect (ci); // refactoring anomaly
9          end
10     end
```

The following structural change matching rules capture Extract Method.

- **Pattern 1**: deleteClone(ci_1, m_1, t_1, r_1) – clone instance ci_1 is deleted from method m_1 in type t_1 in revision r_1.
- **Pattern 2**: addMethod(m_2, t_1, r_1) – method m_2 is added in type t_1 in revision r_1.
- **Pattern 3**: addCall(m_1, m_2, r_1) – a method call to m_2 from method m_1 is added in revision r_1.
- **Pattern 4**: similarBody(ci_1, m_1, m_2, Sim) – the similarity level between a deleted ci_1 in m_1 and a method body of m_2 is greater than threshold Sim.

We believe our approach can be easily extended to support other clone refactorings, which may reuse similar constituent change steps.

Our clone tracking technique for pattern similarBody uses the *Levenshtein distance* algorithm [25], which measures the similarity between two sequences of characters based on number of deletions, insertions, or substitutions required to transform one sequence to the other.

Our approach maps a code snippet $\mathcal{M}_i$ in the clone group and the edited statements $\mathcal{M}_j$ related to changes to $\mathcal{M}_i$ (e.g., clone instances and the body of method `processEvent`

[9]For the interactive and instant inspection, the process of clone filtering of the output of a clone detector is discussed in Section VI.

S	Clone Refactoring Classification
✗	One or more clone instances (CIs) are omitted to apply refactorings in the clone group; however, other clone siblings are refactored.
✗t	CIs use different variable types compared to types of aligned AST nodes in other clone siblings. To handle variations in types, a *parameterize type* refactoring can be considered [30]. The applicability of this refactoring is affected by language support for generic types.
✗m	CIs use different method calls compared to method calls of aligned AST nodes in other clone siblings. To handle variation in method calls, Form Template Method (pg. 345 in [10]) could be applicable by creating common APIs in the base class and encapsulating the variation in the derived classes.
✗r	CIs use different references (e.g., method overriding) compared to references of aligned AST nodes in other clone siblings. Similar to *dynamic-dispatch rule* [6], altering inheritance relations is checked if addition or deletion of an overriding method may result in reference changes.
✗o	CIs are implemented in different orders compared to execution orders in other clone siblings (e.g., `m1();m2();` vs. `m2();m1();`). Form Template Method could be used to allow polymorphism to ensure the different sequences of statements proceed differently.
✗s	Non-syntactic CIs are detected, and syntactic clones expanding clone regions are not similar between clone siblings according to a threshold (Sim). A possible refactoring type is Form Template Method by implementing the details of the different process in the derived classes depending on semantic constraints or the length of common subsequent code.

TABLE II: The six types of classification that PRI annotates classified clone instances with symbols in the first column **S**.

in Figure 1(a)). When comparing $\mathcal{M}_i$ and $\mathcal{M}_j$, our approach generalizes the names of identifiers with abstract variables (e.g., variables, qualifier values, fields, and method parameters) to create a generalized program comparison that does not depend on concrete identifiers. This generalization technique was used in previous works [46], [30], and achieved a high performance to distinguish if two code fragments are relevant. We define the similarity $\mathcal{S}$ between $\mathcal{M}_i$ and $\mathcal{M}_j$ as follows:

$$\mathcal{S} = 1 - \frac{LevenshteinDistance(\mathcal{M}_i, \mathcal{M}_j)}{max(|\mathcal{M}_i|, |\mathcal{M}_j|)} \quad (2)$$

The value of $\mathcal{S}$ is in the interval (0, 1]. Our technique in similarBody searches for a portion of code fragments from method m_2 with a similarity $\mathcal{S}$ of at least a threshold Sim when compared to a clone instance ci_1 deleted in method m_1.

B. Phase II: Detecting Incomplete Clone Refactorings

PRI detects incomplete clone refactorings—there are clone instances which are unrefactored inconsistently with other sibling in the group. It also classifies unrefactored clones if they are locally refactorable or not. Non-locally-refactorable clones means that a developer has difficulty performing refactorings to remove clones using standard refactoring techniques [10] due to limitations of a programming language or incomplete syntactic units of clones.

To detect unrefactored clone instances, we reuse Equation 2. Our approach extracts differences of changes to clones, generalizes statements before and after changes, and computes a distance between $\mathcal{M}_i$ and $\mathcal{M}_j$. If the value of $\mathcal{S}$ is less than Sim, our approach considers $\mathcal{M}_i$ as the unrefactored one and attempts to discover a possible reason why developers have not removed $\mathcal{M}_i$ by a refactoring.

Table II summarizes six types of unrefactored clones which can be automatically classified by PRI. We categorize these cases by manually investigating clone groups from six subject applications used in our case studies (Section V) and plan to add more cases in the future.

Continuing with our example, PRI analyzes an alignment between a pair of two different strings "focusKeyTyped" and "changeEventAction" in Figures 1(a) and (d). It then maps their locations to AST nodes and searches their declarations by performing static interprocedural slicing [15] to determine if their types share a common type. PRI annotates with symbol **✗t** a clone instance in Figure 1(d), which is not factored out unlike other clone siblings due to the type variation.[10] Figure 5 shows how PRI reports detection results to help developers inspect clones for refactorings. A_1, A_2, B_1 and B_2 in Figures 1(a) and (b) are summarized as refactorings; however, C and D in Figures 1(c) and (d) are detected as anomalies.

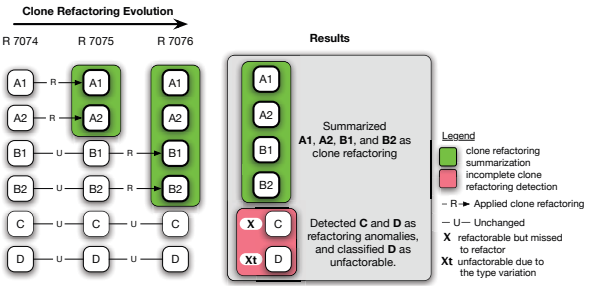


Fig. 5: Detecting refactoring anomalies in clone refactoring evolution.

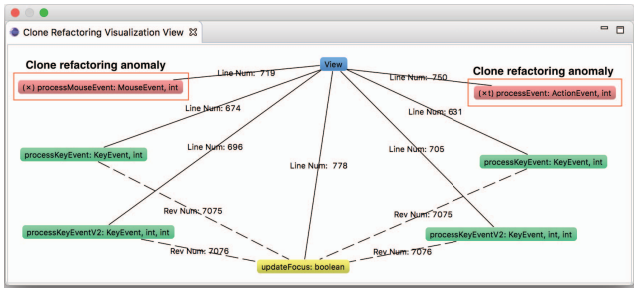


Fig. 6: Visualizing clone refactorings and anomalies regarding Extract Method (see more screenshots at http://faculty.ist.unomaha.edu/msong/pri/).

C. Phase III: Visualizing Clone Refactorings and Anomalies

As PRI is intended for interactive use, we have implemented it in the context of the Java editor of Eclipse, a widely used extensible open-source development environment for Java. PRI shows the *clone refactoring visualization view* that graphically represents refactoring edits and anomalies in a clone group in a tree graph view. Continuing with our example, Figure 6 shows a snapshot of this view. We represent the structural relationship in a clone group with a solid line denoting their locations (i.e., line number); the class `View` (blue) contains six clone instances. We also represent the reference dependence with a

[10]Generic types could be considered to remove the clone, but Fowler's catalog does not include such techniques and current refactoring engines, such as Eclipse, do not support it,

dotted line indicating refactoring histories (i.e., revision); the four clone instances (green) refactored in `processKeyEvent` and `processKeyEventV2` are linked to method `updateFocus` (yellow). The two unrefactored methods (red) are marked as refactoring anomalies without a link to the extracted method `updateFocus`.

V. Evaluation

For assessing PRI's effectiveness, we performed two studies. We first assessed PRI's summarization of clone refactorings and applicability in real scenarios using six open source projects by mining clone refactorings from their repositories. In the second study, we used a data set with real refactoring anomalies—incomplete clone refactorings to investigate its detection capability.

To guide our investigation, we defined the following research questions:

- **RQ1. Can PRI accurately *summarize* clone refactorings?**
- **RQ2. Can PRI accurately *detect* incomplete clone refactorings?**

The next sections discuss each study and its results in detail.

A. Experimental Design

To evaluate PRI, we collected the data set by manually examining clone groups and their changes where real developers applied clone refactorings in repositories. Table III shows details of six subject applications used in our evaluation. We selected these projects for two main reasons. First, all subject applications are written in Java, which is one of the most popular programming languages.[11] Second, these applications are under active development and are based on a collaborative work with at least 48 months of active change history.

Application	Description	File	LOC
AlgoUML	UML modelling tool	1,559	127,145
Apache Tomcat	Web Application server	1,537	215,584
Apache Log4j	Java-based logging utility	817	59,499
Eclipse AspectJ	Aspect-oriented extension to Java	4,758	326,563
JEdit	Java text editor	561	107,368
JRuby	Java implementation of Ruby	1,256	186,514

TABLE III: Subject applications (**File**: the number of files, and **LOC**: lines of code)

This experiment was conducted on a machine with a quad-core 2.2GHz CPU and 16GB RAM.

Data Set. To measure PRI's capability, we established a ground truth set from six subject applications in Table III in the following steps. First, we parsed commit logs to a bag-of-words and stemmed the bag-of-words using a standard NLP tool [29]. We then used keyword matching (e.g., "duplicated code" and "refactoring") in the stemmed bag-of-words to find corresponding revisions. Based on these revisions, we manually investigated changed files to find clone refactorings that real developers performed or missed one or more clone instance in the same clone group. Table IV shows data sets we randomly selected to create as a ground truth set.

[11] http://www.tiobe.com/tiobe-index/

To measure PRI's capability to track the clone evolution, we manually decomposed refactorings into individual changes and committed these changes across revisions to our evaluation repository. For example, we committed n_1 revisions, when Extract Method was applied to clone instances in n_1 methods, and we committed n_2 revisions, when Pull Up Method was applied to clone instances in n_2 subclasses, where $n_i \geq 3$ and the inspection revision scope in the third column in Table IV is n_i + 1 including an original version.

Each clone refactoring is a pair of (P, P') of a program, where P is an original version with clone groups, and P' is a new version that factors out clone groups. If refactorings in P' are performed across revisions by completely removing clone groups in P, we add these changes in our data set G_1 to evaluate RQ1. If any clone instance of a clone group in P remains unrefactored in P' within 10 subsequent revisions,[12] we add these incomplete clone refactorings as anomalies in our data set G_2 to evaluate RQ2.

Using the ground truth data set G_1, precision P_1 and recall R_1 are calculated as $P_1 = \frac{|G_1 \cap S|}{|S|}, R_1 = \frac{|G_1 \cap S|}{|G_1|}$, where P_1 is the percentage of our summarization results that are correct, R_1 is the percentage of correct summarization that PRI reports, and S denotes the clone groups identified by PRI, all clone instances of which are refactored. Using the ground truth data set G_2, precision P_2 and recall R_2 are calculated as $P_2 = \frac{|G_2 \cap D|}{|D|}, R_2 = \frac{|G_2 \cap D|}{|G_2|}$, where P_2 is the percentage of our detection results that are correct, R_2 is the percentage of correct detection results that PRI reports, and D indicates the clone groups detected by PRI, some clone instances of which are not refactored. We measure accuracy using F_1 score by calculating a harmonic mean of precision and recall.

B. Study Results and Discussion

Table IV summarizes our evaluation result, to answer the questions raised above. We collect results for PRI and manually assess the outcomes to collect precision, recall, and accuracy. Regarding validation process, the first author analyzed PRI's results. The results then were validated in the meetings with the remaining authors. When there was any disagreement, each issue was put to a second analysis round, and a joint decision was made.

RQ1. Can PRI accurately *summarize* clone refactorings? We assess PRI's precision by examining how many of refactorings of clone groups are indeed true clone refactoring. PRI summarizes 94 refactorings of clone groups, 92 of which are correct, resulting in 98.9% precision. Regarding recall, PRI identifies 92% of all ground truth data sets. It identifies 92 out of 98 refactored groups. It summarizes refactorings of clone groups, tracking the clone histories with 94.1% accuracy.

PRI summarizes refactorings of clone groups that are not easy to identify because they require investigating refactorings of individual versions of a program while tracking changes of a clone group, e.g., a clone instance refactored in revision r_i and its sibling of the clone refactored in revision r_{i+j}. Instead

[12] As 37% of clone genealogies last an average of 9.6 revisions in Kim et al.'s study [20], we chose 10 subsequent revisions.

				Clone Refactoring Summarization					Incomplete Refactoring Detection									
ID	**RFT**	**VER**	**TIM**	**CL**$_s$	**GT**$_s$	**P**	**R**	**A**	**CL**$_d$	**GT**$_d$	**✗**	**✗t**	**✗m**	**✗o**	**✗s**	**P**	**R**	**A**
1	EM	3	1.1	2 / 1	2 / 1	100	100	100	48 / 3	48 / 3	0	0	43	0	5	100	100	100
2	PU	3	0.1	5 / 2	6 / 3	100	66.7	80.0	123 / 52	123 / 52	49	2	8	0	65	98.1	100	99.0
3	PU	4	0.4	6 / 1	7 / 2	100	50.0	66.7	102 / 38	105 / 39	83	0	1	7	14	100	97.4	98.7
4	ES	9	0.9	20 / 4	20 / 4	100	100	100	449 / 36	449 / 36	43	7	0	0	399	100	100	100
5	ES	4	0.4	2 / 1	9 / 3	100	33.3	50.0	318 / 44	322 / 45	109	0	7	0	206	95.7	97.8	96.7
6	ES	3	0.8	17 / 7	17 / 7	100	100	100	1,430 / 86	1,430 / 86	67	162	49	0	1153	98.9	100	99.4
7	ES	3	0.6	48 / 24	48 / 24	100	100	100	1,118 / 99	1,118 / 99	112	15	18	0	973	100	100	100
8	EM+PU	7	1.0	3 / 1	8 / 2	100	50.0	66.7	237 / 50	237 / 50	84	9	17	0	132	96.2	100	98.0
9	EM	3	0.8	4 / 2	4 / 2	100	100	100	20 / 7	20 / 7	7	0	7	0	6	100	100	100
10	EM	9	0.9	8 / 1	8 / 1	100	100	100	2,643 / 103	2,643 / 103	24	6	14	0	2,599	100	100	100
11	PU	4	0.7	9 / 3	9 / 3	100	100	100	2,228 / 218	2,306 / 221	277	20	90	0	1,919	99.1	98.6	98.9
12	PU	3	0.5	2 / 1	2 / 1	100	100	100	127 / 52	133 / 55	34	8	2	0	89	100	94.5	97.2
13	PU	3	1.1	2 / 1	2 / 1	100	100	100	8 / 3	12 / 5	0	0	12	0	0	100	60.0	75.0
14	PU	3	0.3	2 / 1	2 / 1	100	100	100	222 / 66	222 / 66	58	3	6	0	155	100	100	100
15	PU	3	0.4	2 / 1	2 / 1	100	100	100	171 / 57	171 / 57	50	3	6	0	112	100	100	100
16	ES	3	2.3	40 / 20	40 / 20	100	100	100	2,597 / 64	2,597 / 64	78	16	7	0	2,496	100	100	100
17	PU+EM	3	2.2	6 / 3	6 / 3	100	100	100	1,218 / 42	1,218 / 42	15	0	15	0	1,188	100	100	100
18	ES	3	1.4	11 / 5	12 / 6	71.4	83.3	76.9	1,210 / 41	1,212 / 42	16	7	13	0	1,176	95.3	97.6	96.5
19	EM+PU	3	1.5	2 / 1	2 / 1	100	100	100	2 / 1	2 / 1	2	0	0	0	0	100	100	100
20	EM	3	0.6	2 / 1	2 / 1	100	100	100	50 / 15	50 / 15	4	0	0	0	46	100	100	100
21	MN	3	2.4	2 / 1	2 / 1	100	100	100	2 / 1	2 / 1	2	0	0	0	0	100	100	100
22	PU	3	0.1	2 / 1	2 / 1	100	100	100	29 / 12	29 / 12	4	2	0	0	23	100	100	100
23	PU	3	0.5	2 / 1	2 / 1	100	100	100	97 / 32	99 / 33	14	9	2	0	74	100	97.0	98.5
24	EM	3	0.6	2 / 1	2 / 1	100	100	100	23 / 11	23 / 11	8	3	0	0	12	100	100	100
25	EM	3	0.6	2 / 1	2 / 1	100	100	100	15 / 7	15 / 7	6	3	0	0	6	100	100	100
26	ES	4	0.7	6 / 3	6 / 3	100	100	100	0 / 0	0 / 0	-	-	-	-	-	-	-	-
27	EM+MM	5	1.3	8 / 3	8 / 3	100	100	100	24 / 12	24 / 12	8	0	2	0	14	100	100	100
TOTAL or AVG.		103	0.9	217 / 92	232 / 98	98.9	92.0	94.1	14,511 / 1,152	14,610 / 1,164	1,154	275	319	7	12,862	99.4	97.8	98.4

TABLE IV: Accuracy of PRI's summarization and detection. **RFT**: the refactoring types that developers perform across **VER** revisions (see Table I for acronyms), **VER**: the number of revisions where developers apply refactorings, **TIM**: the time that PRI completes each task (an average of time (sec.) per group), **CL**$_s$: the number of clones correctly summarized by PRI (instance/group), **GT**$_s$: the number of the ground truth data set for clone refactoring summarization (instance/group), **CL**$_d$: the number of clones correctly detected by PRI (instance/group), **GT**$_d$: the number of the ground truth data set for incomplete clone refactoring detection (instance/group), **✗**, **✗t**, **✗m**, **✗o**, and **✗s**: see Table II, **P**: precision (%), **R**: recall (%), and **A**: accuracy (%).

of tracking each version incrementally, simply comparing with the latest version can produce every change to be inspected. However, *composite* code changes, which intermingle multiple development issues together, are commonly difficult to conduct a code review [42].

False Positives. One group is a false positive due to the partitioning issue in clone groups. For example, Extract Super Class is applied to a clone group in Apache Tomcat (r1742245). Although PRI identifies a refactored clone group, the group is not mapped to the ground truth as a clone detector detects the group as two separate groups. Selective merging analysis of clone instances of clone groups can prevent this false positive, which will be included in our heuristics in the future.

False Negatives. One group is not identified by PRI; the refactoring can only be identified by decreasing a similarity level in a clone detector. As PRI relies on the output of a clone detector, it is impossible to identify the refactoring of a group until a clone detector can find all sibling clones. In ArgoUML (r11784), we observed that changing the default setting from 0.95% to 0.85% can let PRI summarize refactorings of the group not reported when using the default setting.

Our threshold Sim also prevents PRI from identifying one group, since the reported cloned regions deviate from the regions that a refactoring is applied in real scenarios. For example, Extract Method is applied to smaller portions than the reported clones in ArgoUML (r16118). This false negative also negatively affects detection of incomplete clone refactorings with respect to precision since refactored clones, which PRI is unable to identify, is considered as unrefactored clones. Controlling Sim can allow to capture such issues.

Other groups not identified by PRI are modified after refactoring application in ArgoUML. We manually investigate the changes to the clone groups and find that refactoring edits includes extra edits without preserving the behavior after applying refactorings to clone groups. Checking the correctness of refactorings to determine whether extra edits are added to the pure refactoring version is our future work.

RQ2. Can PRI accurately *detect* incomplete clone refactorings? We estimate the precision of PRI by evaluating how many of the unrefactored clones are indeed a true omission of refactoring. As PRI detects clone groups in which some clone instances are omitted from refactorings either intentionally or unintentionally, we consider any instance resulting in refactoring deviations of other refactored siblings in the group as a true clone refactoring anomaly. PRI detects 1,162 unrefactored groups, 1,152 of which are correct, resulting in 99.4% precision. Regarding recall, PRI detects 97.8% of all ground truth data sets. It detects 1,152 out of 1,164 unrefactored groups with clone refactoring classification with 98.4% accuracy.

PRI helps developers investigate unrefactored clone instances and understand how these instances are diverged from other clone siblings. It automatically classifies whether unrefactored clone instances cannot be easily removed by

```
class AbstractAjpProtocol {
  void pause() {
    try { ...
    endpoint.pause();
    ... } catch { ... }
  }
  void resume(byte[] data) {
    try { ...
    endpoint.resume();
    ... } catch { ... }
  }
  void stop(byte[] data) {
    try { ...
    endpoint.stop();
    ... } catch { ... }
  }
}
```

```
class AbstractHttp11Protocol {
  void pause() {
    try { ...
    endpoint.pause();
    ... } catch { ... }
  }
  void resume(byte[] data) {
    try { ...
    endpoint.resume();
    ... } catch { ... }
  }
  void stop(byte[] data) {
    try { ...
    endpoint.stop();
    ... } catch { ... }
  }
}
```

Fig. 7: A false negative example from the Apache Tomcat (r1042872) project (the regions with highlighted background are cloned).

standard refactoring techniques [10], which is not easy to determine since understanding clone differences between clone instances in the group usually requires both mapping aligned statements line by line and checking if these statements are implemented with the same program elements (e.g., types or method calls).

False Positives. Most groups are incorrectly classified due to the lack of capability to find functionally identical code clones in the clone detector we used. We investigate the implementations of such clone groups and find that these groups can be reorganized by common functionality. For example, CI1, CI2, CI3, and CI4 are reported in the same group. A pair of CI1 and CI3 and a pair of CI2 and CI4 share the same features, respectively, but the former pair has the type variation and the latter one has the method call variation. Each reorganized group may be classified depending on the variations between counterparts. This limitation can be overcome by plugging in clone detectors that are more resilient to differences in syntax [21], [11].

False Negatives. We inspected the groups not detected by PRI. We found that some clone instances in a group can be reorganized as a sub-group, and other clone siblings can be moved to another sub-group, which causes false negatives. Figure 7 exemplifies the false negatives. As a clone detector reports six clone instances as a group, PRI tracks the change history and detects the group as unrefactored. However, partitioning based on program semantics can allow PRI to detect three unrefactored sub-groups: a set of `AbstractAjpProtocol.pause` and `AbstractHttp11Protocol.pause`, a set of `AbstractAjpProtocol.resume` and `AbstractHttp11Protocol.resume`, and a set of `AbstractAjpProtocol.stop` and `AbstractHttp11Protocol.stop`. These missing groups are hard to detect using PRI since our current templates are not capable of partitioning or merging clone groups to detect refactorable subgroups or most common groups. Heuristics to consider the scenario are planned as future work.

Discussion. PRI classifies 88.0% of unrefactored clones as unfactorable due to involving non-syntactic clone instances and lack of common code. These clone instances break a syntactic boundary, which is expanded to valid ASTs during the classification analysis, but a lower similarity ($< Sim$) is achieved due to lack of common code. For example, syntactic clones expanded by PRI consist of different numbers of statements from other clone siblings, which either co-evolve, diverge, or remain unchanged. Program dependency analysis and heuristics could be used for rearranging unrelated variant code and combining extractable code to remove the majority of clones that PRI reports as unfactorable. However, simultaneous editing [31] may be required to support clone evolution if resulting refactored code would have poor readability. We believe that PRI provides the consistent refactoring information about clones, and automatically places concerns to watch for inconsistent changes to clones for helping developers determine refactoring desirability.

PRI classifies 4.1% of unrefactored clones as a variation of types or method calls. These incomplete refactorings occur due to limited language support for generic types or lack of common code.

PRI classifies seven unrefactored clones as different orders of API calls, which require a consistent sequence of API calls before applying refactorings.

To preserve inheritance relationships between clone instances in the group, we implement a checker to inspect clones creating different references (e.g., method overriding), but we do not observe such cases in our data sets.

The clone refactoring detection algorithm we present in Section IV-A is parameterized with a threshold Sim that determines how closely two code regions must match to be considered equivalent regions. For this threshold, we determine an appropriate value by empirically measuring the success of the clone refactoring summarization for different threshold levels. We randomly select 14 clone groups consisting of more than two clone instances from subject applications in Table III. For each data set, we run PRI to search for refactored regions across revisions. We repeat this experiment for seven different values of Sim between 0.35 and 0.95.

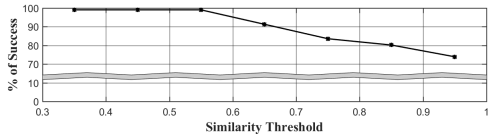


Fig. 8: Impact of similarity threshold on success.

Figure 8 shows the impact of the threshold on the results. The lower the threshold, the higher the success level, because a greater number of code fragments are matched. This trend flattens with a threshold of 0.55, which results in a success level of 98.9%. Overall, to parameterize our algorithm, we determine an appropriate value with 0.55 as a working threshold based on this experiment.

VI. THREATS TO VALIDITY

Our evaluation with only open source projects that are implemented in Java may not generalize to projects. Further investigation is required to validate PRI on projects that are developed with different settings, such as programming languages, application domains, or development organizations. The definition of unfactorable clones in our clone classification depends on the Java programming language, which may not be applicable to other programming languages.

VII. Conclusion

In this paper, we present PRI to analyze how clone instances are refactored consistently (or inconsistently) with other siblings in the same group. To summarize clone refactorings and detect incomplete refactoring anomalies, PRI employs refactoring pattern templates and traces cloned code fragments across revisions. It further analyzes refactoring anomalies to classify if developers are not easy to remove them using standard refactoring techniques. It also provides a novel visualization tool to highlight refactoring edit histories and anomalies. The evaluation shows that our static analysis approach can effectively identify clone refactorings with 94.1% accuracy and detect refactoring anomalies with 98.4% accuracy from six open source projects. Its capability helps developers focus their attention on refactorings to clone groups across versions of a program. As future work for improving PRI's approach and tool, we intend to (i) create pattern templates for more refactoring types; (ii) provide tool support for fixing incomplete clone refactorings; and (iii) implement checking operations to determine the correctness of extra edits to pure refactoring versions. We also plan to conduct user studies with both student and professional developers to improve PRI's usability.

References

[1] L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *Proc. of CSMR*, pages 81–90. IEEE, 2007.

[2] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *Proc. of ICSM*, pages 273–282. IEEE, 2011.

[3] M. Barnett, C. Bird, J. Brunet, and S. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proc. of ICSE*, pages 134–144. IEEE, May 2015.

[4] G. Bavota, A. De Lucia, A. Marcus, R. Oliveto, and F. Palomba. Supporting extract class refactoring in eclipse: The aries project. In *ProcC of ICSE*, pages 1419–1422. IEEE, 2012.

[5] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *Proc. of WCRE*, pages 85–94. IEEE, 2009.

[6] O. C. Chesley, X. Ren, and B. G. Ryder. Crisp: A debugging tool for Java programs. In *Proc. ICSM*, pages 401–410. IEEE, 2005.

[7] E. Duala-Ekoko and M. P. Robillard. Tracking code clones in evolving software. In *Proc. of ICSE*, pages 158–167. IEEE, 2007.

[8] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):18, November 2007.

[9] S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Proc. of ICSE*, pages 222–232, 2012.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[11] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proc. ICSE*, pages 321–330, 2008.

[12] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *Proc. of ICSE*, pages 211–221. IEEE, 2012.

[13] N. Gode and J. Harder. Clone stability. In *Proc. of CSMR*, pages 65–74. IEEE, 2011.

[14] N. Göde and R. Koschke. Studying clone evolution using incremental clone detection. *Journal of Software: Evolution and Process*, 25(2):165–192, 2013.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *Journal of TPLS*, 12(1):26–60, 1990.

[16] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an environment for the proactive management of copy-and-paste programming. In *Proc. ICPC*, pages 238–242. IEEE, 2009.

[17] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proc. of ICSE*, pages 96–105. IEEE, 2007.

[18] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proc. of ESEC-FSE*, pages 55–64. ACM, 2007.

[19] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proc. of ICSE*, pages 151–160. ACM, 2011.

[20] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proc. of ESEC/FSE*, pages 187–196, 2005.

[21] J. Krinke. Identifying similar code with program dependence graphs. In *Proc. of WCRE*, page 301. IEEE, 2001.

[22] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Proc. of WCRE*, pages 170–178. IEEE, 2007.

[23] J. Krinke. Is cloned code more stable than non-cloned code? In *Proc. of SCAM*, pages 57–66. IEEE, 2008.

[24] J. Krinke. Is cloned code older than non-cloned code? In *Proc. of IWSC*, pages 28–33. ACM, 2011.

[25] V. I. Levenstein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10*, 10(8):707–710, 1966.

[26] J. Li and M. D. Ernst. CBCD: Cloned buggy code detector. In *Proc. of ICSE*, pages 310–320. IEEE, 2012.

[27] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proc. of OSDI*, pages 289–302, 2004.

[28] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao. Detecting differences across multiple instances of code clones. In *Proc. of ICSE*, pages 164–174. ACM, 2014.

[29] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60, 2014.

[30] N. Meng, L. Hua, M. Kim, and K. S. McKinley. Does automated refactoring obviate systematic editing? In *Proc. of ICSE*, pages 392–402. IEEE, 2015.

[31] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–174, Berkeley, CA, USA, 2001. USENIX Association.

[32] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proc. of SAC*, pages 1227–1234. ACM, 2012.

[33] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Elipse IDE? *Software, IEEE*, 23(4):76–83, 2006.

[34] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proc. of ASE*, pages 180–190, 2013.

[35] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Clone-aware configuration management. In *Proc of ASE*, 2009.

[36] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *Proc. of MSR*, pages 40–49, 2012.

[37] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Detection of recurring software vulnerabilities. In *Proc. of ASE*, pages 447–456. ACM, 2010.

[38] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *Proc. of ICSM*, pages 1 –10, 2010.

[39] B. Ray, M. Kim, S. Person, and N. Rungta. Detecting and characterizing semantic inconsistencies in ported code. In *Proc. of ASE*, pages 367–377. IEEE, 2013.

[40] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[41] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry. Understanding the evolution of type-3 clones: an exploratory study. In *Proc. of MSR*, pages 139–148. IEEE, 2013.

[42] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proc. of FSE*, 2012.

[43] M. Toomim, A. Begel, and S. L. Graham. Managing duplicated code with linked editing. In *Proc. VLHCC*, pages 173–180. IEEE, 2004.

[44] N. Tsantalis and A. Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[45] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proc. of ICSE*, pages 233 –243, 2012.

[46] T. Zhang, M. Song, J. Pinedo, and M. Kim. Interactive code review for systematic changes. In *Proc. of ICSE*, pages 111–122, 2015.