Received: 3 August 2017 | Revised: 11 February 2018 | Accepted: 23 February 2018
DOI: 10.1002/smr.1951

RESEARCH ARTICLE - STANDARDS

WILEY Journal of Software: Evolution and Process

Clone refactoring inspection by summarizing clone refactorings and detecting inconsistent changes during software evolution

Zhiyuan Chen[1] | Young-Woo Kwon[2] | Myoungkyu Song[1]

[1] University of Nebraska, Omaha, NE, USA
[2] Kyungpook National University, Daegu, South Korea

Correspondence
Myoungkyu Song, Computer Science Department, University of Nebraska at Omaha.
Email: myoungkyu@unomaha.edu

It has been broadly assumed that removing code clones by refactorings would solve the problems of code duplication. Despite recent empirical studies on the benefit of refactorings, contradicting evidence shows that it is often difficult or impossible to remove clones by using standard refactoring techniques. Developers cannot easily determine which clones can be refactored or how they should be maintained scattered throughout a large code base in evolving systems. We propose pattern-based clone refactoring inspection (PRI), a technique for managing clone refactorings. PRI summarizes refactorings of clones and detects clones that are not consistently refactored. To help developers refactor these anomalies, PRI also visualizes clone evolution and refactorings and fixes refactoring anomalies to prevent the clone group from being left in an inconsistent state. We evaluated PRI on 6 open-source projects and showed that it identifies clone refactorings with 94.1% accuracy and detects inconsistent refactorings with 98.4% accuracy, tracking clone change histories. In a study with 10 student developers, the participants reported that flexible PRI's summarization and detection features can be valuable for novice developers to learn about refactorings to clones. These results show that PRI should improve developer productivity in inspecting clone refactorings distributed across multiple files in evolving systems.

KEYWORDS
code clone, refactoring, software maintenance and evolution

1 | INTRODUCTION

Code clones undergo repetitive similar edits.[1,2] Previous research efforts focused on similar code changes find that up to 30% of the total amount of code are clones in various types of applications,[3,4] resulting in negative impacts on software quality, such as hidden bug propagation and unintentional inconsistent changes. It is widely believed that code clones are inherently harmful and that refactoring can solve these clones to improve software quality and developer productivity.

Refactorings are behavior-preserving program transformations that improve the design of a program. Opdyke and Johnson[5] introduced the term in 1990, and Fowler[6] popularized it 9 years later. Examples of refactorings include extracting duplicated code into a new method (Extract Method) or moving a group of similar methods to a superclass (Pull Up Method). Each refactoring comprises a set of preconditions and specific transformation patterns to perform.[6]

Several prior research efforts provide positive effects of code clones, mostly because of an easy method to achieve a design goal by the reuse of the same design patterns or libraries.[7-9] According to Kim et al's study,[10] developers are often able to discover a shared abstraction of clones after they keep and maintain clones for some period time before they realize how to refactor the common part of the clones. On the other hand, according to other empirical studies, cloning is considered harmful and may result in poor software quality.[11-13] For example, developers could introduce bugs when changing the code if they mistakenly ignore to change related clones.[12,13] They could inadvertently repeat anomalous changes made by their own errors or by others' fault.[11]

To cope with numerous issues related to clones, clone detection tools have been proposed using text-based,[14] token-based,[15] tree-based,[16] and graph-based[17] approaches. These techniques find clones by matching code fragments with gaps, demonstrating to scale in large software systems. Once clones are found, refactoring can remove identified clones.[6,18,19]

Although several clone detectors have been proposed,[20] it is still challenging for developers to analyze clone evolution patterns to understand and manage a clone group containing multiple clone instances as it evolves. In clone management, developers frequently apply clone removal refactorings to clone groups, manually investigating the clone evolution over time. Modern integrated development environments (IDEs), including Eclipse, IntelliJ IDEA, NetBeans, and Visual Studio, have provided automated refactoring tools to make refactoring tasks more efficient and reliable.

Nevertheless, recent studies find that most developers underuse automated refactorings, and more often perform refactorings manually.[21-25] Developers know that refactoring tools are available in IDEs, but they often do not know how to configure these automated refactoring tools and do not trust them.[25] Professional developers also tend not to use automated refactoring despite their awareness of the refactoring features of IDEs owing to usability issues or a lack of trust.[25] Murphy et al find that 90% of refactorings are manually applied without automated refactoring tools.[22] Negara et al observe that most expert developers prefer manual refactorings over automated tool support for refactorings.[24] These manual refactorings are often prone to errors.

Several studies find that refactorings are closely related to bug fixes regarding *time* and *location* where developers make changes.[26] Weißgerber and Diehl find that a refactoring rate is strongly related to a bug fix rate.[27] Dig and Johnson report that over 80% of application programming interface (API)–breaking changes are caused by the refactoring task, leading to errors in the existing applications.[28] Kim et al conduct a field study at Microsoft and find that 77% of developers consider refactoring as an activity with a risk of introducing subtle refactoring bugs and functionality regression.[21] As a result, it is very likely to be risks for developers to refactor clone groups, since they are left to manually investigate individual, evolving clone instances in the clone group to answer questions such as “How should these clones be refactored completely or correctly?” and “Are there any other clones that should be removed along with these clones?”

To address these problems, we present a technique, pattern-based clone refactoring inspection (PRI), for efficiently managing clone refactorings. To answer the aforementioned questions, PRI (1) summarizes clone refactorings and (2) detects inconsistent or incomplete refactoring edits. To better support these summarization and detection tasks, it is designed as an Eclipse IDE plug-in[29] for visualizing clone evolutions and fixing refactoring anomalies. We target 5 refactoring types that are the most commonly performed in practice[22,25] (Extract Method, Pull Up Method, Move Method, Extract Super Class, and Move Type to New File) and 2 composite refactorings (Extract & Move Method and Extract & Pull Up Method). For example, PRI summarizes a set of the evolving clone groups, including their refactored revisions and associated change information such as corresponding refactoring types, locations, and restructuring descriptions. To provide an insight of clone evolution to both novice and skilled developers, PRI classifies unrefactored clones: (1) refactorable clones along with a recommended refactoring type and (2) unfactorable clones along with a discovered reason of not being refactorable.

We evaluated PRI on over 100 versions of 6 open-source projects. We mined a ground truth data set by manually examining the changed source code and the corresponding commit logs to find clone refactorings that real developers either completely or inconsistently refactored in one or more clone siblings in the same clone group. The data set was constructed without compilation errors since PRI relies on reference binding analysis on syntactically valid abstract syntax trees (ASTs), instead of containing only change history snapshots. On the basis of this ground truth data set, we conducted 2 case studies and a study with student developers. First, we evaluated the accuracy of PRI's refactoring summarization by comparing PRI's results against the ground truth and found that it shows a precision of 98.9% and a recall of 92%. In our second case study, we evaluated PRI on a data set with real-world clone refactoring anomalies to which real developers applied incomplete refactorings and found that it detects such refactoring anomalies with 99.4% precision and 97.8% recall by automatically tracking evolving clones and the corresponding refactoring edits across revisions. We also conduct a user study with 10 computer science students, in which they found PRI useful for inspecting clone refactorings. The participants also provided strong positive responses that they would like to use PRI to detect and locate refactoring anomalies rather than use existing refactoring tools.

In summary, our paper makes the following contributions:

- PRI, a novel static analysis approach and an open-source implementation to inspect evolving clones for refactoring. The analysis approach includes program differencing and AST-based code pattern search. This is the first work to track the refactoring changes to clones for summarizing clone refactorings and for detecting each unrefactored clone instance or its group. It extracts clone differences and classifies a set of unrefactored clones—which either coevolved, diverged, or remain unchanged—to provide relevant hints to help answer what, how, and why these clones are not refactorable.
- An algorithm and implementation to support developers with a summarization of clone refactorings and detection of incomplete refactorings of clone groups as potential anomalies. Our technique is designed in an open-source plug-in for the Eclipse IDE that shows the refactoring edit history on a visualization view. Also, it provides tool support for fixing incomplete or inconsistent refactoring edits by systematically disabling the preconditions of a refactoring engine that leads to the rejection of the transformation.
- An empirical evaluation showing the correctness and usefulness of PRI in the clone refactoring management. The evaluation includes 2 case studies with over 100 revisions of 6 open-source projects to assess PRI's (1) summarization capability for refactoring changes to evolving clones and (2) the detection capability for potential anomalies of incomplete refactorings. Also, our user study shows PRI helps developers inspect clone refactorings and detect refactoring anomalies by providing useful feedback.

The rest of this article is organized as follows. Section 2 reviews the related work. Section 3 demonstrates our approach using a motivating example. Section 4 describes how our static program analysis matches refactoring templates using change patterns and how our tool supports the clone evolution visualization and the refactoring anomaly removal. Section 5 empirically evaluates our approach. After discussing the limitation of our approach in Section 6, we conclude and outline future work directions in Section 7.

2 | RELATED WORK

2.1 | Modern code review tools for clone refactorings

Existing code review tools, such as Codeflow (https://goo.gl/xaE4Pe), Collaborator (https://goo.gl/oBz3ea), Gerrit (http://code.google.com/p/gerrit/), and Phabricator (http://phabricator.org), 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.

Recent work on refactoring-aware code reviews addresses the problem of manually conducting refactorings and proposes a code review tool that detects behavior changes in refactoring edits.[30-32] However, PRI focuses on managing clone refactorings of clone groups and the consistent evolution and detecting incomplete refactorings as potential anomalies.

2.2 | Detecting inconsistent (or consistent) changes to clones

Several approaches detect inconsistencies in clones. CP-Miner,[33] SecureSync,[34] and Deckard[12] reveal clone-related bugs by finding recurring vulnerable codes. CBCD reveals bug propagation in clones by finding similar ASTs in a program dependence graph.[35] SPA analyzes discrepancies in changes and detects inconsistent updates in clones.[36]

Our approach differs from these inconsistency detection techniques in 2 ways. First, PRI automatically accesses version control systems (VCSs) and incrementally identifies inconsistent changes in clone histories, unlike analysis of 1 or 2 versions. Because of these differences, in our case studies, we could not directly compare it 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.[6]

Göde and Koshke[37] present an incremental clone detection algorithm to study clone evolution and find that most clones remain unchanged during their lifetime or clones are mostly changed inconsistently. Saha et al[38] 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. Bettenburg et al[39] investigate the effect of inconsistent changes to code clones and observe that 1% to 3% of inconsistent changes to clones introduce defects. There are also contradicting studies for consistent changes to clones. Krinke's[8,40] study investigates the changes to clones in 5 open-source systems and finds that some clone groups are changed consistently. Aversano et al[41] study how clones are evolved and find that developers almost propagate the change consistently. 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 or beneficial. PRI helps summarize clone evolution and refactorings, which developers can investigate during peer code reviews.

Toomim et al,[42] Duala-Ekoko and Robillard,[43] Hou et al,[44] Nguyen et al,[45] and Jablonski et al[46] manage clones in evolving software by tracking the changes in the clones as code evolves. Lin et al,[47] Hou et al,[48] and Jacob et al[49] 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 that are omission prone, supporting *clone-aware refactoring recommendations*.

2.3 | Identifying clone refactoring candidates

Tsantalis et al[50] 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[51] 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.

BeneFactor[52] and WitchDoctor[53] 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[54] study the evolution of clones and provide a manual classification for evolving code clones. Unlike their approach, ours automatically classify clones if they are not easily refactorable using standard refactoring techniques.[6]

Mondal et al[55] study clone evolution to find cochange association among clone classes by mining association rules from history of clone changes. Similar to our approach for consistent refactorings, their approach investigates a tendency of changing together during evolution of a clone group. Their technique focuses on removing clone fragments by identifying and ranking candidates for refactoring; however, in addition to the candidate identification, our approach provides clone summarization that helps developers inspect and classify clone evolution across revisions.

Tsantalis et al[56] present an approach that examines the *refactorability* of clones to determine whether duplicated codes can be safely modified without changing the program behavior. Their approach takes as input clones, searches for identical nesting structures, and finds a set of mappings between statements aiming a higher refactorability. Similar to our approach that uses a set of refactoring pattern rules, their approach then uses a set of matching preconditions and examines a set of mapped statements to determine whether they are refactorable with semantics-preserving transformation. However, their approach focuses on a pair of clones (ie, a pair of matched AST subtrees) to check precondition violations. By contrast, our approach analyzes clone groups containing more than 2 clone instances and determines whether each clone group is refactorable by differentiating subtrees among multiple clone instances. Like our technique for the change investigation of clone refactorings, Tairas and Gray's[57] also use the Eclipse refactoring engine for the purpose of clone refactorings.

While their approaches above perform refactorings only if they find no precondition violations, our approach complements these normal refactoring cases; it enables a user to proceed refactorings despite a scenario of refactoring anomalies by refining overly strong preconditions built in the refactoring engine.

Hotta et al[58] present a technique called Creios that applies clone refactoring using program dependence graphs. While their approach focuses on the Form Template Method refactoring, PRI supports 5 refactoring types that are the most commonly performed in practice,[22,25] including 2 composite refactorings. Creios only highlights the candidates, but PRI allows application of refactorings by the systematic interoperation with the refactoring engine of an IDE.

Bian et al[59] present SPAPE, an amorphous procedure refactoring technique, by extracting the noncontiguous lines of clones (type 3) into a method. SPAPE finds the differing statements in near-miss clones on PDGs and merges these statements by inserting control variables and conditional statements on ASTs. It then replaces the near-miss clones with a procedure call. Although SPAPE merges differing statements of the near-miss clone fragments, it decreases code understandability by introducing complex conditional logic and control variables. Unlike PRI completely running within an IDE, the authors emphasize the need for further improvement for the IDE integration.

Tsantalis et al[60] present a clone refactoring technique that uses lambda expressions[61] to parameterize the behavioral differences for refactoring either type 2 or type 3 clones. After the unmapped statements (ie, clone gaps) are associated with *precondition violations* in their prior work,[56] their algorithm determines whether these unmapped statements can be parameterized in the form of lambda expressions. Although their refactoring technique handles behavioral differences between a pair of clones, applying refactorings with lambda expressions may cause version compatibility issues since lambda expressions are introduced after Java 8. We observed that developers often produce refactoring anomalies that inconsistently refactor some clone instances of the clone group (eg, missing to apply Extract Method or Pull Up Method to some clone instances). To overcome the problem and to complement existing approaches, PRI focuses on detecting and repairing the anomalies to help developers consistently complete the same refactoring on all clone instances. For clone instances that PRI detects and classifies as unfactorable (eg, method call variation) with standard refactorings,[6] their technique with lambda expressions is possibly adapted to parameterize behavior differences.

3 | 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 the Extract Method to 2 cloned regions in methods processKeyEvent and processKeyEventV2, respectively. (Method processKeyEventV2 is created to support the different version.) She validates manual refactorings[25] 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 in conducting the same refactoring because of the type variation of the variable changeEventAction, which is different from corresponding variables of other clone siblings in the same group.

To ensure that there is no location Alice missed to refactor, Barry needs to investigate line-level differences file by file during the peer code review. 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.[62,63] Since understanding such composite changes requires nontrivial efforts,[63] subchanges 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, which is the changes she commits, and then he runs PRI, a plug-in built on top of Eclipse IDE. On the basis of r_i, PRI tracks r_i, r_{i+1}, …, r_{i+n} and summarizes clone refactorings performed by Alice (Figure 1A) and applied by other developers reusing the existing code by altering a copy-pasted code (Figure 1B). The results include refactored revisions, types (eg, Extract Method), locations (eg, package, class, method, and line number), and restructuring descriptions (eg, “Method m1 is refactored and cloned code fragments are replaced with a call to an extracted method m2”). PRI also detects unrefactored clones in a clone group (Figure 1C), classifying if a clone instance is locally refactorable by standard refactorings[6] (Figure 1D).

```
public void processKeyEvent(KeyEvent evt, int from) {
   Event focusKeyTyped, focusKeyPressed;
   Event event = (Event) evt;
   ..
   switch(evt.getID()) {
   case Event.KEY_TYPED:
-      if(inputHandler.isActive() && from != VK_CANCEL) {..
-           focusKeyTyped = event.getEvent();
-           ..
-      }
+      focusKeyTyped = processEvent(from, focusKeyTyped, event);
       ..
   case Event.KEY_PRESSED:
       event = (Event) evt;
-      if(inputHandler.isActive() && from != VK_CANCEL) {..
-           focusKeyPressed = event.getEvent();
-           ..
-      }
+      focusPressed = processEvent(from, focusKeyPressed, event);
 ..}
}

+ Event processEvent(int from, Event evt, Event event) {
+  if(inputHandler.isActive() && from != VK_CANCEL) {
+       evt = event.getEvent();
+       ..
+  }
+  return event;
+ }
```

(A)

```
public void processKeyEventV2(KeyEvent evt, int from, int enhancedVer) {
   Event focusKeyTypedEvt, focusKeyPressedEvt;
   Event event = (Event) evt;
   ..
   switch(evt.getID()) {
   case Event.KEY_TYPED:
-      if(inputHandler.isActive() && from != VK_CANCEL) {..
-           focusKeyTypedEvt = event.getEvent();
-           ..
-      }
+      focusKeyTypedEvt = processEvent(from, focusKeyTypedEvt, event);
   ..
   case Event.KEY_PRESSED:
       event = (Event) evt;
-      if(inputHandler.isActive() && from != VK_CANCEL) {..
-           focusKeyPressedEvt = event.getEvent();
-         ..
-      }
+      focusKeyPressedEvt = processEvent(from, focusKeyPressedEvt, event);
 ..}
}
```

(B)

```
void processMouseEvent(MouseEvent ev, int sr) {
   Event focusMousePressed;
   Event event = (Event) ev;
   switch(evt.getID()) {
   case Event.MOUSE_PRESSED:
       if(inputHandler.isActive() &&
                sr != VK_CANCEL) {..
           focusMousePressed = ev.getEvent();
           ..
       }
       ..
   }
}
```

(C)

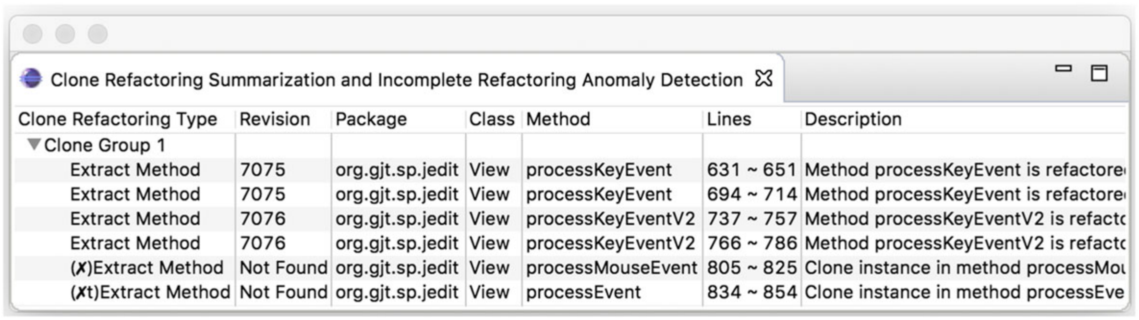
```
void processActionEvent(ActionEvent ev, int sr) {
   Action changeEventAction;
   Action action = (Action) ev;
   switch(evt.getID()) {
   case Event.CTRL_MASK:
       if(inputHandler.isActive() &&
                sr != VK_CANCEL) {..
           changeEventAction = ev.getEvent();
           ..
       }
       ..
   }
}
```

(D)

FIGURE 1 A simplified example: a clone group is refactored inconsistently. Cloned regions are highlighted, deleted code is marked with "−" and added code marked with "+." A, A clone refactoring in revisions v20060919-7074 and v20060919-7075 in JEdit. B, A refactoring to clones of the same group in a later revision by another developer. C, A clone instance where Alice misses to apply the same refactoring as in A and B. D, A clone instance in the clone group, which is not refactorable, unlike A to C. The type of changeEventAction in D differs from aligned variables in other siblings in the clone group.

Figure 2 shows a snapshot of a PRI viewer, which shows a clone group that comprises 6 clone instances searched in each row of a tree viewer. The first 2 rows show the clone instances factored out by the Extract Method in revision 7075, and the next 2 rows show the clone instances refactored in the same way in the next revision 7076 (Section 4.1). PRI marks the fifth clone instance with symbol ***X***, meaning that it detects an omission error (Figure 1C). It marks the last clone instance with symbol ***Xt***, meaning that it discovers Alice's difficulty in applying the Extract Method in the same way because of the type variation (Figure 1D) (Section 4.2). (Java generic types could be applicable for a refactoring, which is not included in standard refactorings.[6]) The viewer in Figure 2 is synchronized with a visualization view (Section 4.3) designed for inspecting clone refactorings regarding structural and dependence relationships between clone instances and related contexts. Using a refactoring tool in PRI (Section 4.4), he also fixes Alice's mistake (Figure 1C) with an awareness of no update to finish such incomplete refactorings in a repository.

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


Clone Refactoring Summarization and Incomplete Refactoring Anomaly Detection

Clone Refactoring Type	Revision	Package	Class	Method	Lines	Description
▼Clone Group 1						
Extract Method	7075	org.gjt.sp.jedit	View	processKeyEvent	631 ~ 651	Method processKeyEvent is refactore
Extract Method	7075	org.gjt.sp.jedit	View	processKeyEvent	694 ~ 714	Method processKeyEvent is refactore
Extract Method	7076	org.gjt.sp.jedit	View	processKeyEventV2	737 ~ 757	Method processKeyEventV2 is refactc
Extract Method	7076	org.gjt.sp.jedit	View	processKeyEventV2	766 ~ 786	Method processKeyEventV2 is refactc
(✗)Extract Method	Not Found	org.gjt.sp.jedit	View	processMouseEvent	805 ~ 825	Clone instance in method processMou
(✗t)Extract Method	Not Found	org.gjt.sp.jedit	View	processEvent	834 ~ 854	Clone instance in method processEve

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

instances; however, revealing these locations and classifying them whether to easily be removed using standard refactoring techniques can be worthwhile during a code review.

4 | PATTERN-BASED CLONE REFACTORING INSPECTION

This section presents PRI consisting of the following 4 phases. Phase 1 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 2 detects clone groups incompletely refactored, classifying the reasons. Phase 3 provides code visualization to represent historical refactoring edits that phase 1 finds and refactoring opportunities that phase 2 detects. Phase 4 provides tool support for fixing clone groups detected in phase 2 via an automated refactoring engine that our approach adapts for correct refactorings. Figure 3 outlines the workflow of PRI.

4.1 | Phase 1: change summarization for clone refactorings

4.1.1 | Converting clone to AST model

Our approach parses clone groups reported by *Deckard*, a clone detector.[16] The default setting used was 30 minT (minimal number of tokens required for clones), 2 stride (size of the sliding window), and 0.95 *Similarity*. We used this default setting because developers do not generally consider very short code snippets as code clones, and many research projects have used this default setting to find code clones. PRI finds the set of AST nodes that contains reported clones:

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

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 innermost syntactic statements (ie, 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 the Java source code and generating the corresponding ASTs is provided with the Eclipse Java Development Toolkit framework (http://www.eclipse.org/jdt/). The AST model helps to conduct reference binding analysis and edit matching operations during AST traversing processes.

4.1.2 | Tracking clone histories

PRI accesses each subsequent revision $r_j \in R = \{r_{i+1}, \ldots, r_n\}$ 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 tools using a software configuration management library (http://www.svnkit.com/) 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 the 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

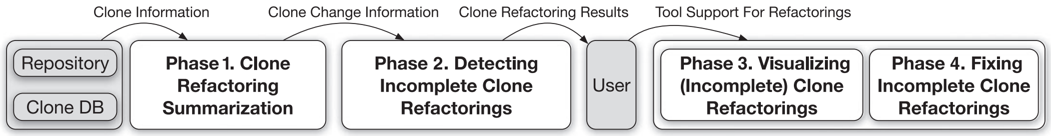

Clone Information
Clone Change Information
Clone Refactoring Results
Tool Support For Refactorings
Repository
Clone DB
Phase 1. Clone Refactoring Summarization
Phase 2. Detecting Incomplete Clone Refactorings
User
Phase 3. Visualizing (Incomplete) Clone Refactorings
Phase 4. Fixing Incomplete Clone Refactorings

FIGURE 3 Overview of PRI's workflow

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 2 revisions.

4.1.3 | Extracting changes between 2 versions of ASTs

PRI leverages ChangeDistiller[64] 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.

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). In Figure 4(A), deleted nodes are highlighted in green with dotted line rectangles. These nodes exist before applying the refactoring change. In Figure 4(B), inserted nodes are highlighted in orange with solid line rectangles. These nodes appear after applying the refactoring change. It then uses ChangeDistiller to compute the differences between 2 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 changes before and after clone refactorings are checked, it is important to determine if references (eg, method call and field access) are preserved across clone instances. Therefore, we create a new reference binding checker, which is not provided by the ChangeDistiller, to assess reference consistency. For 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.

To construct the binding information of ASTs, the data set used in our evaluation (Section 5) contains a set of revisions that constructs syntactically valid ASTs without compilation errors, instead of maintaining only change history snapshots.[65] For example, PRI automatically accesses VCS of the subject application to load each revision in Eclipse IDE. Regarding use case scenarios, refactoring anomalies do not produce compilation errors that developers are likely to overlook for checking these locations.

4.1.4 | Matching clone refactoring pattern templates

Refactoring pattern templates are AST-based implementations that consist of a pair of *pre-edit and postedit matchers*, such as $\mathcal{M}_{\text{pre}}$ and $\mathcal{M}_{\text{post}}$. $\mathcal{M}_{\text{pre}}$ is an implementation for matching patterns before clone refactoring application. $\mathcal{M}_{\text{post}}$ 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 (eg, refactored revisions, refactoring types, locations, and restructuring descriptions). The change pattern descriptions are designed by using declarative rule-based constraints.[66]

Table 1 shows clone refactoring templates that our approach can identify based on pre-edit and postedit matchers. We leverage the rules of refactoring types in Fowler's catalog.[6] A composite refactoring comprises a set of low-level refactorings. For example, template 6 in Table 1 describes that the Extract Method is applied to a clone group, and the new method is moved to another class. Template 7 shows that the Extract Method is applied to a clone group in subclasses whose new methods are moved to their superclasses. Template 8 describes that the Pull Up Method is applied to a clone group whose superclasses are refactored by the Extract Method in moved methods.

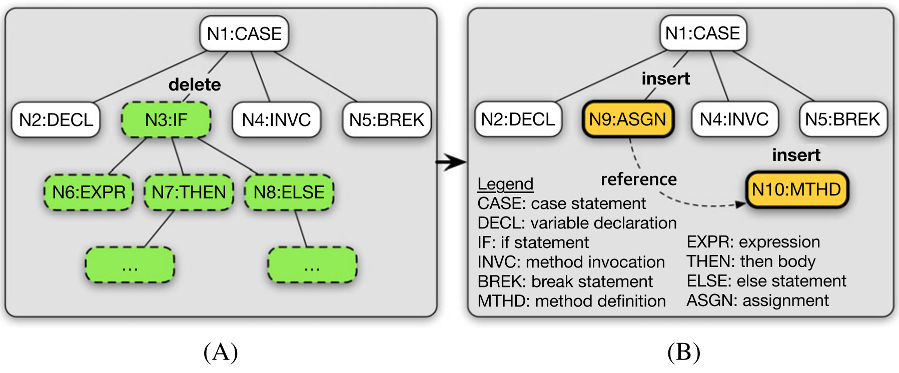


FIGURE 4 Extracting the edit operations from clone changes. A, Before refactoring a clone instance. B, After refactoring a clone instance

TABLE 1 The refactoring pattern templates for capturing structural change matching rules

ID	Template Rule and Description
1 EM	Extract Method refactoring pattern rule
	P1.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.
	P2.addMethod(m_2, t_1, r_1): Method m_2 is added in type t_1 in revision r_1.
	P3.addCall(m_1, m_2, r_1): A method call to m_2 from method m_1 is added in revision r_1.
	P4.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*.
2 MM	Move Method refactoring pattern rule
	P1.deleteCloneMethod(ci_1, m_1, t_1, r_1): Method m_1 as a clone instance ci_1 in the clone group is deleted in type t_1 in revision r_1.
	P2.addMethod(m_2, t_1, r_1): Method m_2 is added in type t_1 in revision r_1.
	P3.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*.
	P4.notEqual(t_1, t_2): Types t_1 and t_2 have different declarations.
3 PM	Pull Up Method refactoring pattern rule
	P1.deleteCloneMethod(ci_1, m_1, t_1, r_1): Method m_1 as a clone instance ci_1 in the clone group is deleted in type t_1 in revision r_1.
	P2.addMethod(m_2, t_2, r_1): Method m_2 is added in type t_1 in revision r_1.
	P3.equal(m_1, m_2): Types t_1 and t_2 have the same declaration.
	P4.subtype(t_1, t_2): Type t_1 is a subtype of t_2.
4 ES	Extract Superclass refactoring pattern rule
	P1.deleteCloneMethod(ci_1, m_1, t_1, r_1): Method m_1 as a clone instance ci_1 in the clone group is deleted in type t_1 in revision r_1.
	P2.addType(t_2, r_1): Type t_2 is added in revision r_1.
	P3.addSubtype(t_1, t_2, r_1): Type t_1 is added into a set of subtypes of type t_2 in revision r_1.
	P4.addMethod(m_2, t_2, r_1): Method m_2 is added in type t_2 in revision r_1.
	P5.equal(m_1, m_2): Types t_1 and t_2 have the same declaration.
5 MN	Move Type to New File refactoring pattern rule
	P1.deleteCloneType(ci_1, t_1, t_2, r_1): Type t_1 as a clone instance ci_1 in the clone group is deleted in type t_2 in revision r_1.
	P2.addType(t_3, p_1, r_1): Type t_3 is added in package p_1 in revision r_1.
	P3.equal(t_1, t_3): Types t_1 and t_3 have the same declaration.
6 EM+MM	Extract and Move Method refactoring pattern rule
	P1.deleteClone(ci_1, m_1, t_1, r_1): A clone instance ci_1 of the group is deleted from method m_1 in type t_1 in revision r_1.
	P2.addMethod(m_2, t_2, r_1): Method m_2 is added in type t_1 in revision r_1.
	P3.addCall(m_1, m_2, r_1): A method call to m_2 from method m_1 is added in revision r_1.
	P4.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*.
	P5.notEqual(t_1, t_2): Types t_1 and t_2 have different declarations.
7 EM+PM	Extract and Pull Up Method refactoring pattern rule
	P1.deleteClone(ci_1,m_1,t_1,r_1): A clone instance ci_1 is deleted from method m_1 in type t_1 in revision r_1.
	P2.addMethod(m_2,t_2,r_1): Method m_2 is added in type t_1 in revision r_1.
	P3.addCall(m_1,m_2,r_1): A method call to m_2 from method m_1 is added in revision r_1.
	P4.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*.
	P5.subtype(t_1,t_2): Type t_1 is a subtype of t_2.
8 PM+EM	Pull Up and Extract Method refactoring pattern rule
	P1.deleteCloneMethod(ci_1, m_1, t_1, r_1): Method m_1 as a clone instance ci_1 in the clone group is deleted from type t_1 in revision r_1.
	P2.addMethod(m_2, t_2, r_1): Method m_2 is added in type t_1 in revision r_1.
	P3.addMethod(m_3, t_2, r_1): Method m_3 is added in type t_1 in revision r_1.
	P4.addCall(m_2, m_3, r_1): A method call to m_3 from method m_2 is added in revision r_1.
	P5.similarBody(ci_1, m_1, m_3, *Sim*): The similarity level between a deleted ci_1 in m_1 and a method body of m_3 is greater than threshold *Sim*.
	P6.subtype(t_1, t_2): Type t_1 is a subtype of t_2.

The single and composite refactorings are included in IDs 1-5 and IDs 6-8, respectively.

```
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      $CGs \leftarrow \mathcal{M}_{\text{pre}}.match(OCD, PRG)$;
3      foreach $Revisionr_i \in REVs$ do
4          foreach $CloneGroup\ G_{\text{clone}} \in CGs$ do
5              $RES \leftarrow \mathcal{M}_{\text{post}}.match(G_{\text{clone}}, r_i)$;
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 (eg, packages, classes, methods, and fields) and structural dependencies (eg, containment, subtyping, overriding, and method calls). M_{pre}*.match*, a matching function, 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 a common superclass (line 2). It then iterates 2 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. 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 matching function 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 (Figure 1A,B) in each revision using templates. During this inspection, Algorithm 2 finds the pattern of Extract Method by performing rules in lines 3 and 4. The structural change matching rule in Table 1 captures the Extract Method, including other refactoring patterns rules. We believe our approach can be easily extended to support other clone refactorings, which may reuse similar constituent change steps.

```
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 $ci \in G$ do
3          if $deleteClone(ci, m_i, t_i, R) \wedge addMethod(m_j, t_i, R) \wedge$
4          $addCall(m_i, m_j, R) \wedge similarBody(m_i, m_j, Sim)$ then
5              $RES \leftarrow summarize(ci)$; // clone refactoring
6          end
7          else
8              $RES \leftarrow detect(ci)$; // refactoring anomaly
9          end
10     end
```

Our clone tracking technique for pattern similarBody uses the *Levenshtein distance* algorithm,[67] which measures the similarity between 2 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 $\mathcal{M}_i$ (eg, clone instances and the body of method processEvent in Figure 1A). When comparing $\mathcal{M}_i$ and $\mathcal{M}_j$, our approach generalizes the names of identifiers with abstract variables (eg, 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[68,70] and achieved a high performance in distinguishing if 2 code fragments are relevant. We define the similarity $\mathcal{S}$ between $\mathcal{M}_i$ and $\mathcal{M}_j$ as follows:

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

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 with a clone instance ci_1 deleted in method m_1.

4.2 | Phase 2: detecting incomplete clone refactorings

PRI detects incomplete clone refactorings—there are clone instances that are unrefactored inconsistently with other sibling in the group. It also classifies unrefactored clones if they are locally refactorable or not. Nonlocally refactorable clones mean that a developer has difficulty performing refactorings to remove clones using standard refactoring techniques[6] owing 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.

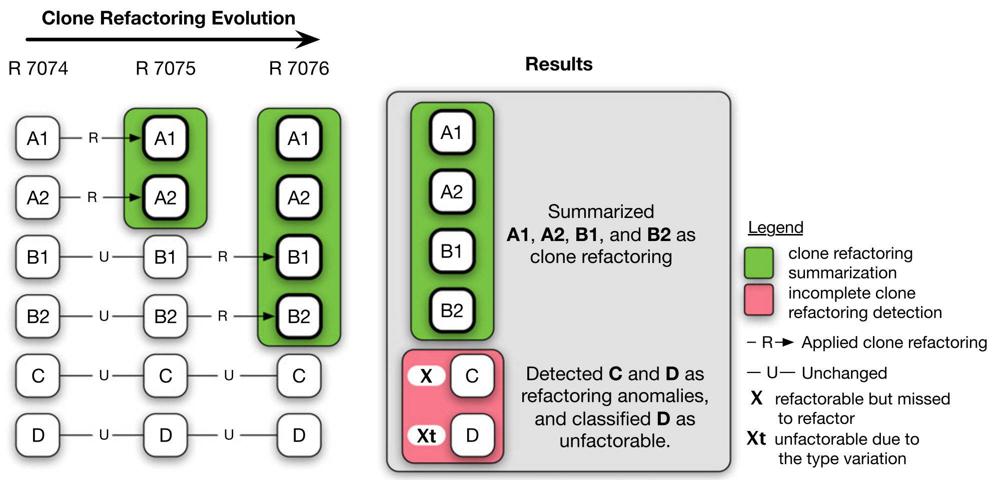
Table 2 summarizes 6 types of unrefactored clones that can be automatically classified by PRI. We categorize these cases by manually investigating clone groups from 6 subject applications used in our case studies (Section 5) and plan to add more cases in the future.

Continuing with our example, PRI analyzes an alignment between a pair of 2 different strings "focusKeyTyped" and "changeEventAction" in Figure 1 (A,D). It then maps their locations to AST nodes and searches their declarations by performing static interprocedural slicing[71] 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 owing to the type variation. (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.) 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 Figure 1(A,B) are summarized as refactorings; however, C and D in Figure 1(C,D) are detected as anomalies.

To annotate anomalies with the symbol **✗m**, PRI analyzes an alignment between a pair of 2 statements among clone instances and detects any different method call compared with method calls of aligned AST nodes in other clone instances in the clone group. PRI applies the *Levenshtein*

TABLE 2 The 6 types of classification in which PRI annotates classified clone instances, with symbols in the first column

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 with types of aligned abstract syntax tree (AST) nodes in other clone siblings. To handle variations in types, a *parameterize type* refactoring can be considered.[68] The applicability of this refactoring is affected by language support for generic types.
✗m	CIs use different method calls compared with method calls of aligned AST nodes in other clone siblings. To handle variation in method calls, Form Template Method[6(p345)] could be applicable by creating common application programming interfaces in the base class and encapsulating the variation in the derived classes.
✗r	CIs use different references (eg, method overriding) compared with references of aligned AST nodes in other clone siblings. Similar to the *dynamic-dispatch rule*,[69] 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 with execution orders in other clone siblings (eg, m1();m2(); vs m2();m1();). The Form Template Method could be used to allow polymorphism to ensure the different sequences of statements proceed differently.
✗s	Nonsyntactic 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 the Form Template Method by implementing the details of the different processes in the derived classes depending on semantic constraints or the length of common subsequent codes.



Clone Refactoring Evolution
R 7074
R 7075
R 7076
A1
A2
B1
B2
C
D
R
U
Results
Summarized A1, A2, B1, and B2 as clone refactoring
X
Xt
Detected C and D as refactoring anomalies, and classified D as unfactorable.
Legend
clone refactoring summarization
incomplete clone refactoring detection
R Applied clone refactoring
U Unchanged
X refactorable but missed to refactor
Xt unfactorable due to the type variation

FIGURE 5 Detecting refactoring anomalies in clone refactoring evolution

distance algorithm[67] for the clone instance alignment by computing the similarity between 2 statements. To annotate anomalies with the symbol **✗r**, PRI checks reference bindings between aligned AST nodes such as method calls or field access among clone instances and detects any different reference binding due to the addition or deletion of overriding methods or fields. To annotate anomalies with the symbol **✗o**, PRI examines an ordering of matching AST nodes between clone instances and detects a set of the statements that can be executed differently, such as intertwining nodes, compared with the execution order in other clone instances. For the **✗s** annotation for anomalies, PRI detects clone instances that can be expanded to the valid syntactic boundary during the AST converting process and detects any different clone instance by computing the similarity between corresponding ASTs of clone instances using a threshold *Sim*. For anomaly annotations with the symbol **✗**, PRI tracks the refactoring changes to a clone group for identifying clone refactorings in Table 1 and detects any clone siblings of the group that are possibly refactorable using standard refactoring techniques[6] in Table 1 except unfactorable clones due to the described reasons in Table 2.

4.3 | Phase 3: 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 (ie, line number); the class View (blue) contains 6 clone instances. We also represent the reference dependence with a dotted line indicating refactoring histories (ie, revision); the 4 clone instances (green) refactored in processKeyEvent and processKeyEventV2 are linked to method updateFocus (yellow). The 2 unrefactored methods (red) are marked as refactoring anomalies without a link to the extracted method updateFocus.

4.4 | Phase 4: fixing incomplete clone refactorings

PRI helps developers examine in isolation incomplete clone refactorings—transformations where clone instances are refactored but other siblings in the same clone group are unrefactored inconsistently. To fix clones that are incompletely refactored, it goes a step further by adapting refactoring features of an existing refactoring engine such as Eclipse (eclipse.org). Refactoring engines may have overly weak and overly strong preconditions.[72-74] Overly weak preconditions ignore incorrect transformation to be applied, while overly strong preconditions reject to apply behavior preserving transformation. We discuss below that *existing refactoring engines have overly strong preconditions when applying refactorings to incomplete clone refactorings*. These issues are not desirable for developers during refactoring tasks. Vakilian and Johnson find that most developers prefer IDEs' refactoring applications without overly strong preconditions,[75] despite manual corrections for some problems (eg, compilation error) after transformation. PRI helps developers perform refactorings on incompletely refactored clones by automatically loading refactoring condition checkers at runtime. It (1) disables overly strong preconditions that existing refactoring engines reject transformation on incompletely refactored clones and (2) applies a behavior-preserving transformation by leveraging automated refactoring features in refactoring engines. Figure 7 shows exemplary implementations where developers easily add new checkers by implementing our predefined interfaces.

4.4.1 | Challenges of fixing incomplete clone refactorings in modern IDEs

To apply refactorings to incompletely refactored clones, a developer may use an existing feature of Eclipse IDE. For example, she applies the Extract Method to the clone in Figure 1(C) that PRI reports as refactorable in phase 2. She runs the Eclipse refactoring tool in Figure 8, selecting

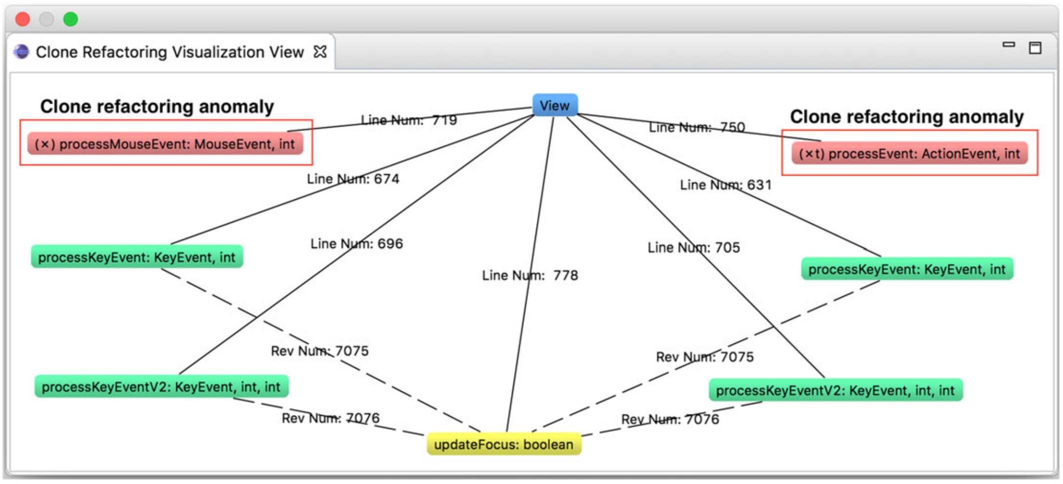

Clone Refactoring Visualization View
Clone refactoring anomaly
(×) processMouseEvent: MouseEvent, int
Clone refactoring anomaly
(×t) processEvent: ActionEvent, int
View
Line Num: 719
Line Num: 750
Line Num: 674
Line Num: 631
Line Num: 696
Line Num: 705
Line Num: 778
processKeyEvent: KeyEvent, int
processKeyEvent: KeyEvent, int
Rev Num: 7075
Rev Num: 7075
processKeyEventV2: KeyEvent, int, int
processKeyEventV2: KeyEvent, int, int
Rev Num: 7076
Rev Num: 7076
updateFocus: boolean

FIGURE 6 Visualizing clone refactorings and anomalies regarding Extract Method (see more screenshots at https://sites.google.com/unomaha.edu/clonerefactoring)

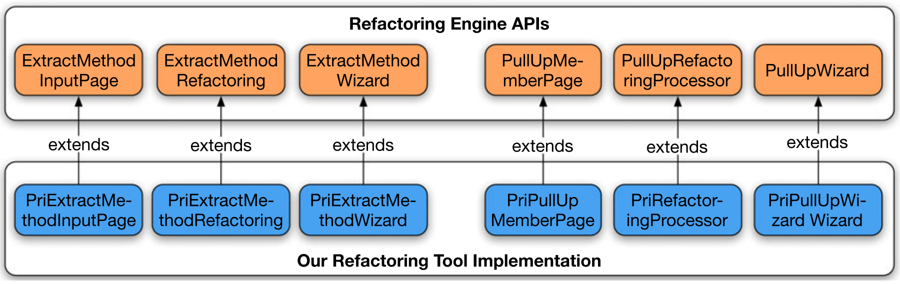
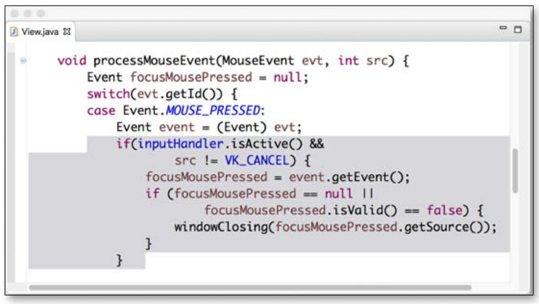
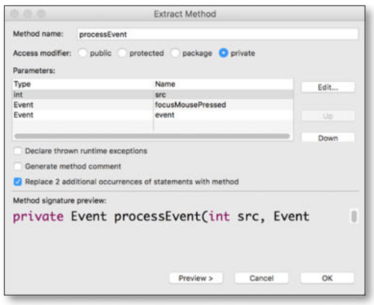


FIGURE 7 PRI's class diagrams that extend existing refactoring application programming interfaces (APIs). Each class implemented in PRI overrides existing checking and abstract syntax tree transformation functionality in Eclipse refactoring APIs



```
void processMouseEvent(MouseEvent evt, int src) {
    Event focusMousePressed = null;
    switch(evt.getId()) {
    case Event.MOUSE_PRESSED:
        Event event = (Event) evt;
        if(inputHandler.isActive() &&
                src != VK_CANCEL) {
            focusMousePressed = event.getEvent();
            if (focusMousePressed == null ||
                    focusMousePressed.isValid() == false) {
                windowClosing(focusMousePressed.getSource());
            }
        }
```

(1) Select a clone instance to fix the incomplete clone refactoring



Method name: processEvent

Access modifier: public protected package [✓ private]

Parameters:

Type	Name
int	src
Event	focusMousePressed
Event	event

☐ Declare thrown runtime exceptions

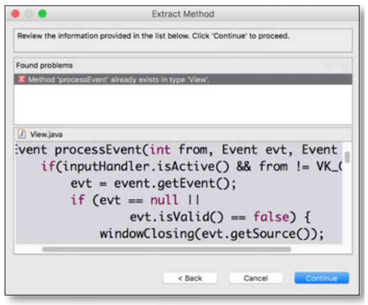
☐ Generate method comment

☑ Replace 2 additional occurrences of statements with method

Method signature preview:

```
private Event processEvent(int src, Event
```

(2) Configure a method name with the same signature processEvent



Review the information provided in the list below. Click 'Continue' to proceed.

Found problems

Method 'processEvent' already exists in type 'View'.

View.java

```
:vent processEvent(int from, Event evt, Event
    if(inputHandler.isActive() && from != VK_(
        evt = event.getEvent();
        if (evt == null ||
                evt.isValid() == false) {
            windowClosing(evt.getSource());
```

(3) A warning message with overly strong conditions

FIGURE 8 A refactoring warning checking overly strong preconditions in Eclipse IDE when the Extract Method is applied to a clone instance in method processMouseEvent in Figure 1 that is incompletely unrefactored unlike other siblings in the same clone group. Step 1 allows a developer to select an extracting location to be refactored by an Extract Method refactoring. Step 2 configures a method name with the same signature as processEvent. Step 3 produces a refactoring warning message "Method processEvent already exists in type View." Continuing to execute the tool creates a duplicate method, causing compilation errors

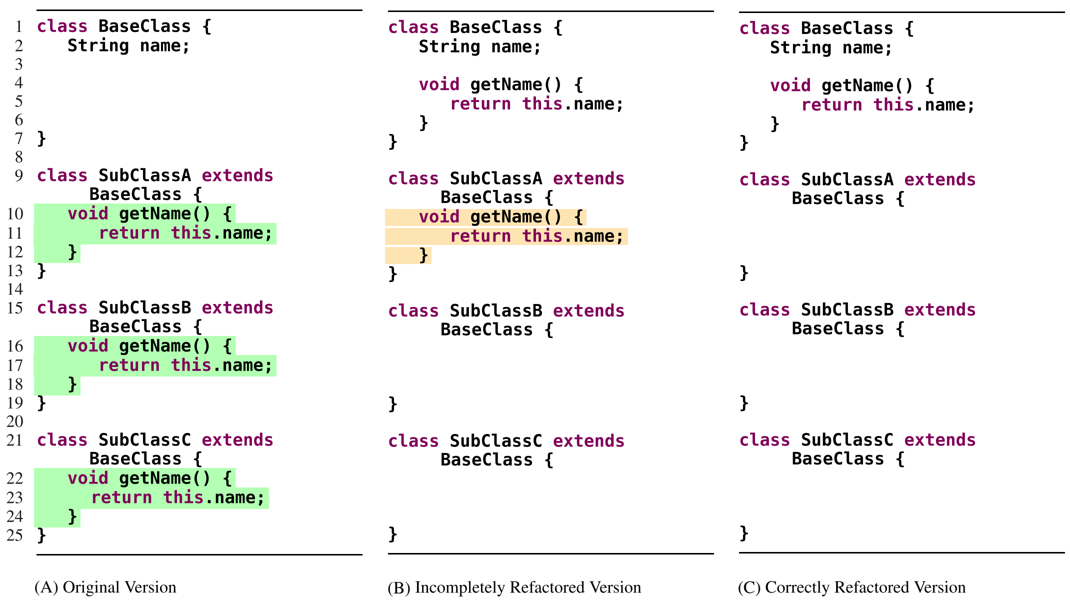
a clone instance in step 1 and configuring an extracting location with a method signature processEvent called from other refactored methods in step 2. However, Eclipse raises a warning message, "Method processEvent already exists in type View" in step 3, which implements overly strong preconditions because the refactoring engine in Eclipse rejects the application of a refactoring (ie, Extract Method) to refactoring anomalies in Figure 1(C), different from other refactored clone siblings in the clone group. If she decides to perform this execution, she must manually fix compilation errors, since Eclipse applies the Extract Method with an addition of the duplicated method. This phenomenon similarly happens in other refactoring types such as the Pull Up Method and in other IDEs such as IntelliJ IDEA (jetbrains.com) and NetBeans (netbeans.org) (https://sites.google.com/a/unomaha.edu/software-eng-research/crc/compare).

Figure 9 shows another simple example wherein a developer finds a clone group from *n* number of subclasses (simplified with 3 subclasses for presentation purposes). She applies the Pull Up Method to the clone group but forgets to perform a refactoring on class SubClassA. To fix such omission anomalies, the refactoring feature in NetBeans IDE can be used to finish a missing clone refactoring in class SubClassA, expecting a refactored version in the right part of Figure 9. NetBeans IDE interacts with a user to remove an omission anomaly in class SubClassA, while reusing the same method getName() moved from classes SubClassB and SubClassC to the superclass BaseClass that was modified by a prior refactoring activity with the Pull Up Method. However, it shows a refactoring warning, "Member getName already exists in the target type," in which overly strong preconditions are implemented, not allowing the execution of the button "Refactor" in Figure 10. It requires conducting of individual refactorings manually, leading to tedious modification.

IntelliJ IDEA and Eclipse suffer from similar problems due to overly strong conditions in Figure 10. If developers using these IDEs proceed with the Pull Up Method on the method SubClassA.getName(), the refactoring engine produces syntactic violations due to duplicated method generation. To correct these errors, they must conduct a manual inspection of unforeseen transformation caused by the tool. Although these issues might not be bugs in the refactoring engines, producing warnings or compilation errors in modern refactoring features due to overly strong conditions hinders the use of automated refactoring tools.[25,72-74] This process for manually localizing and fixing incomplete clone refactorings can be tedious and error prone.

4.4.2 | Applying refactorings to incompletely refactored clones

To complement these modern refactoring tools, we present tool support for fixing potential missed refactoring edits—*omission anomalies* caused by incomplete refactorings of clone groups. (We do not attempt to replace existing refactoring tools, which can still be available.) To design the



```
class BaseClass {
   String name;




}

class SubClassA extends
      BaseClass {
   void getName() {
      return this.name;
   }
}

class SubClassB extends
      BaseClass {
   void getName() {
      return this.name;
   }
}

class SubClassC extends
      BaseClass {
   void getName() {
      return this.name;
   }
}
```

(A) Original Version

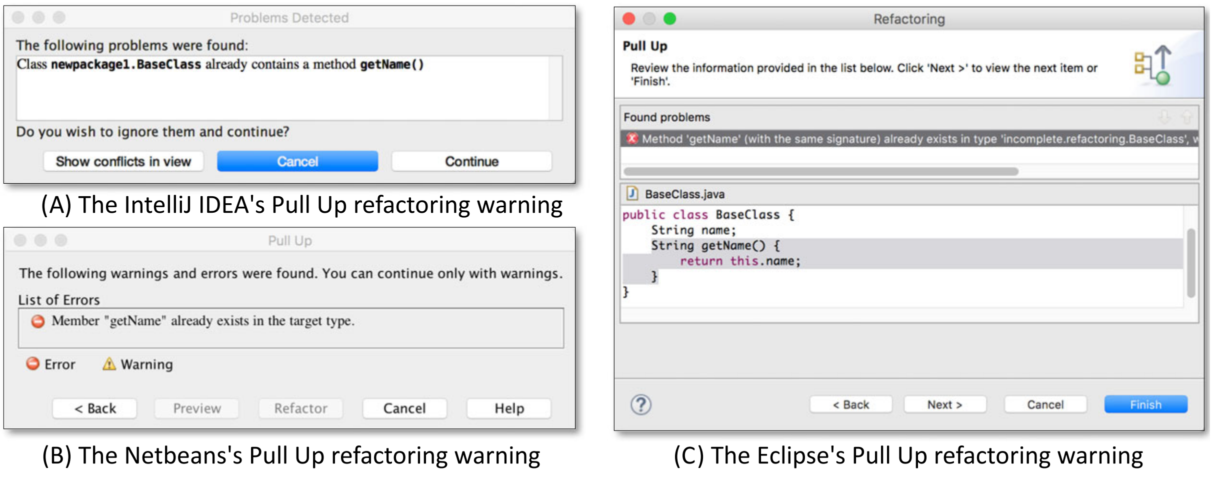
```
class BaseClass {
   String name;

   void getName() {
      return this.name;
   }
}

class SubClassA extends
      BaseClass {
   void getName() {
      return this.name;
   }
}

class SubClassB extends
      BaseClass {


}

class SubClassC extends
      BaseClass {


}
```

(B) Incompletely Refactored Version

```
class BaseClass {
   String name;

   void getName() {
      return this.name;
   }
}

class SubClassA extends
      BaseClass {


}

class SubClassB extends
      BaseClass {


}

class SubClassC extends
      BaseClass {


}
```

(C) Correctly Refactored Version

FIGURE 9 An omission error in the clone group when conducting the Pull Up Method. A, Original version containing the clone group. Cloned regions are highlighted in green. B, Incompletely refactored version that indicates an omission error in the method SubClassA.getName(). The anomaly is highlighted in orange. C, Correctly refactored version



Problems Detected

The following problems were found:
Class newpackage1.BaseClass already contains a method getName()

Do you wish to ignore them and continue?

(A) The IntelliJ IDEA's Pull Up refactoring warning

Pull Up

The following warnings and errors were found. You can continue only with warnings.

List of Errors
Member "getName" already exists in the target type.

(B) The Netbeans's Pull Up refactoring warning

Refactoring

Pull Up
Review the information provided in the list below. Click 'Next >' to view the next item or 'Finish'.

Found problems
Method 'getName' (with the same signature) already exists in type 'incomplete.refactoring.BaseClass', w

BaseClass.java
```
public class BaseClass {
    String name;
    String getName() {
        return this.name;
    }
}
```

(C) The Eclipse's Pull Up refactoring warning

FIGURE 10 Each warning message “Class BaseClass already contains a method getName(),” “Member getName already exists in the target type,” and “Method getName (with the same signature) already exists in type BaseClass, which will result in compiler error if you proceed” with overly strong conditions from refactoring engines in existing IDEs such as IntelliJ IDEA (A), NetBeans (B), and Eclipse (C) when attempting to fix an incomplete refactoring by the Pull Up Method in Figure 9

tool implementation in PRI, we first randomly select 120 (in total 1164) clone groups associated with incomplete clone refactorings from the data set shown in Table 4. Next, the Eclipse refactoring engine performs a standard refactoring[6] such as the Extract Method and Pull Up Method on a clone instance that is missed but where other clone siblings are refactored. When it rejects a transformation, we collect refactoring warnings reported by this engine. For each warning message, we then inspect the refactoring implementation code and identify code fragments related to the overly strong preconditions generating the warning messages that do not allow the resolution of incomplete clone refactorings. The goal is to apply the refactoring on these omission errors instead of producing the warning message to help avoid unsafe manual refactorings.

In phase 2 in Section 4.2, PRI reports that one or more missed clone instances (ie, omission anomalies) are detected to be refactorable in the same clone group. It then (1) disables the refactoring condition checkers that define refactoring preconditions, if the refactoring engine rejects transformations that resolve refactoring omission errors, and (2) prevents the unexpected AST transformations that cause syntactic violations such as compilation errors. For example, if the Extract Method is applied to clone instances but omission anomalies are detected in other sibling(s) in the clone group, PRI fixes these omission anomalies by (1) disabling the preconditions checking whether a method with the same method signature already exists in a type and (2) preventing the engine from applying AST transformation with a duplicate method creation in the same type. As another example, if the Pull Up Method is applied to clone instances but other clone siblings are detected as refactoring omission anomalies, PRI resolves these anomalies by (1) disabling preconditions that lead the engine to check if a superclass already defines a method with the same method signature and (2) preventing AST transformation with a duplicate method generation in the superclass.

We overview PRI's workflow on incomplete clone refactorings in Figure 11. Also, we show how PRI adapts refactoring features to systematically disable overly strong preconditions and disallow AST transformations in Figure 7. We currently support 2 refactoring types, namely, the Extract Method and Pull Up Method, to safely conduct refactorings on clone instances that cause omission anomalies. In the future, we plan to support other refactorings by leveraging refactoring features provided by Eclipse's Java Development Toolkit and Eclipse Language Toolkit.

A previous study finds that most developers rarely use refactoring tools in practice, although these tools refactor more correctly than developers do.[23] Other studies reflect on reasons for this underuse, such as usability issues,[76] unawareness,[25] and a lack of trust.[25,73,77,78] Because of their complexity and error-proneness, refactoring edits require a lot of attention during maintenance and inspection tasks. To relieve the developers from having to check the correctness of refactorings, PRI automatically detects incomplete clone refactorings and allows developers to remove these omission anomalies safely by systematically leveraging automated features of a refactoring engine.

5 | EVALUATION

We evaluated PRI using 2 different methods. First, for assessing PRI's effectiveness, we conducted an exploratory study to evaluate PRI's summarization of clone refactorings and applicability in real scenarios using 6 open-source projects by mining clone refactorings from their repositories. Also, we used a data set with real refactoring anomalies—incomplete clone refactorings—to investigate its detection capability. To assess PRI's capacity of helping developers to investigate clone refactorings and to guide our investigation, we defined the following research questions, and Sections 5.1 and 5.2 discuss each study and its results in detail:

- Can PRI accurately *summarize* clone refactorings?
- Can PRI accurately *detect* incomplete clone refactorings?

Second, we conducted a user study with computer science students to understand how PRI can help them during refactorings. This study demonstrates the realistic refactoring scenarios and solicits authentic feedback on the use of PRI in the real world. Section 5.3 discusses the study result in detail. These 2 different evaluation methods complement each other by assessing the benefits of PRI both qualitatively and quantitatively.

5.1 | Experimental design for case study

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 3 shows details of 6 subject applications used in our evaluation. We selected these projects for 2 main reasons. First, all subject applications are written in Java, which is one of the most popular programming languages (http://www.tiobe.com/tiobe-index/). Second, these applications are under active development and are based on a collaborative work with at least 48 months of active change history.

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

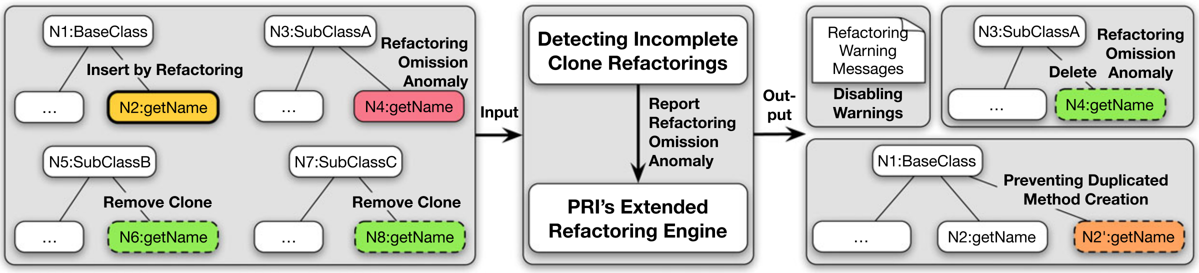


FIGURE 11 Fixing a refactoring omission anomaly in a detected clone instance at the abstract syntax tree node N4 caused by incomplete clone refactorings in Figure 9

TABLE 3 Subject applications

Application	Description	File	LOC
AlgoUML	UML modelling tool	1559	127 145
Apache Tomcat	Web Application server	1537	215 584
Apache Log4j	Java-based logging utility	817	59 499
Eclipse AspectJ	Aspect-oriented extension to Java	4758	326 563
JEdit	Java text editor	561	107 368
JRuby	Java implementation of Ruby	1256	186 514

File: the number of files; LOC: lines of code.

TABLE 4 Accuracy of PRI's summarization and detection

				Clone Refactoring Summarization					Incomplete Refactoring Detection									
ID	RFT	VER	TIM	CL_s	GT_s	P	R	A	CL_d	GT_d	χ	χ_t	χ_m	χ_o	χ_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	1430/86	1430/86	67	162	49	0	1153	98.9	100	99.4
7	ES	3	0.6	48/24	48/24	100	100	100	1118/99	1118/99	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	2643/103	2643/103	24	6	14	0	2599	100	100	100
11	PU	4	0.7	9/3	9/3	100	100	100	2228/218	2306/221	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	2597/64	2597/64	78	16	7	0	2496	100	100	100
17	PU+EM	3	2.2	6/3	6/3	100	100	100	1218/42	1218/42	15	0	15	0	1188	100	100	100
18	ES	3	1.4	11/5	12/6	71.4	83.3	76.9	1210/41	1212/42	16	7	13	0	1176	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 average		103	0.9	217 / 92	232 / 98	98.9	92.0	94.1	14 511/1152	14 610/1164	1154	275	319	7	12 862	99.4	97.8	98.4

Abbreviations: A, accuracy (%); CL_d, the number of clones correctly detected by PRI (instance/group); CL_s, the number of clones correctly summarized by PRI (instance/group); GT_d, the number of ground truth data sets for incomplete clone refactoring detection (instance/group); GT_s, the number of the ground truth data set for clone refactoring summarization (instance/group); ID, each data set that contains the refactoring changes across multiple revisions; P, precision (%); R, recall (%); RFT, the refactoring types that developers perform across VER revisions (see Table 1 for acronyms); TIM, the time during which PRI completes each task for both clone refactoring summarization and incomplete refactoring detection (an average of time [s] per group); VER, the number of revisions where developers apply refactorings. For χ, χ_t, χ_m, χ_o, and χ_s, see Table 2.

5.1.1 | Data set

To measure PRI's capability, we established a ground truth data set from 6 subject applications in Table 3 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.[79] We then used keyword matching (eg, "duplicated code" and "refactoring") in the stemmed bag-of-words in commit messages to find corresponding revisions in which developers intended to perform refactorings to remove clones. On the basis of these revisions, we manually investigated changed files to find clone refactorings that real developers performed or where they missed one or more clone instances in the same clone group. However, our observation in Section 6 discusses that commit messages often do not reliably specify the refactoring changes. Table 4 shows data sets we randomly selected to create as a ground truth set.

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 number of revisions when the Extract Method was applied to clone instances in n number of methods. Similarly, we committed m number of revisions when the Pull Up Method was applied to clone instances in m number of subclasses, where n(or m) ≥ 3 and the inspection revision scope in the third column in Table 4 is $n + 1$, including an original version. For example, the data set in ID 1 includes the Extract Method (EM) that is performed across 3 revisions (VER). It simulates a use case of the clone evolution as follows: a developer applies the Extract Method (RFT) to one clone instance of the group (GT_s) in the first revision leading, to the production of the second revision. Then she subsequently evolves the same clone group of the second revision to complete the same refactoring for the other clone siblings, resulting in the last revision. It also simulates a use case of the incomplete clone evolution as follows: a developer does not apply the Extract Method to 3 clone groups including 48 clone instances (GT_d), which are unrefactored clones due to the method call variation (**✗m**) and the nonsyntactic clone instances (**✗s**).

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, we add these incomplete clone refactorings as anomalies in our data set G_2 to evaluate RQ2. (As 37% of clone genealogies last an average of 9.6 revisions in Kim et al's study,[54] we chose 10 subsequent revisions.)

Using the ground truth data set G_1, precision P_1 and recall R_1 are calculated as $P_1 = |G_1 \cap S|/|S|$ and $R_1 = |G_1 \cap S|/|G_1|$, 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 $P_2 = |G_2 \cap D|/|D|$ and $R_2 = |G_2 \cap D|/|G_2|$, 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.

5.1.2 | Impact of similarity threshold

The clone refactoring detection algorithm we present in Section 4.1 is parameterized with a threshold *Sim* that determines how closely 2 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 2 clone instances from subject applications in Table 3. For each data set, we run PRI to search for refactored regions across revisions. We repeat this experiment for 7 different values of *Sim* between 0.35 and 0.95.

Figure 12 shows the impact of the threshold on the results. With a high threshold *Sim* of 0.95, clone refactoring summarization was not very successful at 73.9%. This phenomenon is not surprising because the very rigid similarity requirement indicates that equivalent code regions cannot be identified because very few clones are exact. 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%. This flattening demonstrates the robustness of PRI's algorithm, because our algorithm generalizes identifiers (eg, variables, qualifier values, fields, and method parameters) before comparing 2 code regions. We do not measure the success level for lower threshold values, because these values are likely not to have further impact on the results and the experiment is very labor-intensive. Overall, to parameterize our algorithm, we determine an appropriate value of 0.55 as a working threshold based on this experiment.

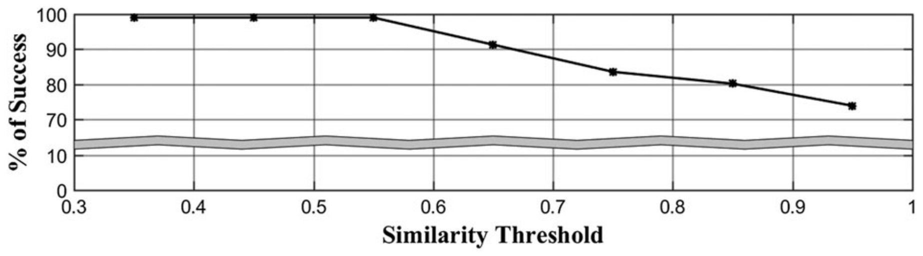


FIGURE 12 Impact of similarity threshold on success

5.2 | Case study results and discussion

Table 4 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.

5.2.1 | RQ1: can PRI accurately *summarize* clone refactorings?

We assess PRI's precision by examining how many refactorings of the clone groups are indeed true clone refactorings. PRI summarizes 94 refactorings of clone groups, 92 of which are correct, resulting in a 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, eg, a clone instance refactored in revision r_i and its sibling of the clone refactored in revision r_{i+j}. Instead of tracking each version incrementally, a simple comparison with the latest version can cause every change to be inspected. However, it is commonly difficult to conduct a code review in *composite* code changes, which intermingle multiple development issues together.[63]

False positives

One group is a false positive due to the partitioning issue in clone groups. For example, the 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 since a clone detector detects the group as 2 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 OCD, 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 to which a refactoring is applied in real scenarios. For example, the 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, are considered as unrefactored clones. Controlling *Sim* can allow the capture of 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 include 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.

5.2.2 | 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 1162 unrefactored groups, 1152 of which are correct, resulting in 99.4% precision. Regarding recall, PRI detects 97.8% of all ground truth data sets. It detects 1152 out of 1164 unrefactored groups with clone refactoring classification with 98.4% accuracy.

PRI helps developers investigate unrefactored clone instances and understand how these instances diverged from other clone siblings. It automatically classifies whether unrefactored clone instances cannot be easily removed by standard refactoring techniques,[6] which is not easy to determine by using existing code review tools such as Codeflow, Collaborator, Gerrit, and Phabricator. On the basis of these tools typically used in practice, understanding clone differences between clone instances in the group usually requires developers both to map aligned statements line by line and to check if these statements are implemented with the same program elements (eg, types or method calls).

False positives

Most groups are incorrectly classified owing 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.[17,80]

False negatives

We inspected the groups not detected by PRI. We found that some clone instances in a group can be reorganized as a subgroup, and other clone siblings can be moved to another subgroup, which causes false negatives. Figure 13 exemplifies the false negatives. As a clone detector reports 6

```
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 { ... }
 }
}
```

FIGURE 13 A false-negative example from the Apache Tomcat (r1042872) project (the regions with highlighted background are cloned)

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 3 unrefactored subgroups: 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.

5.2.3 | Discussion

PRI classifies 88.0% of unrefactored clones as unfactorable owing to involving nonsyntactic 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 because of a lack of common code. For example, syntactic clones expanded by PRI consist of different numbers of statements from other clone siblings, which coevolve, 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[81] may be required to support clone evolution if the resulting refactored code would have poor readability. We believe that PRI provides consistent refactoring information about clones and automatically places concerns to watch for inconsistent changes to clones for helping developers determine refactoring desirability. In addition, the above result is a ripe opportunity for code refactoring researchers to have an impact on real-world scenarios as the majority of clones are known type 2 and type 3 clones. To handle replacement of a method call with different code in each clone, we may use a refactoring technique,[59] SPAPE, that performs amorphous transformation on PDGs by inserting control variables and conditional statements on ASTs to replace the near-miss clones with a method call, although SPAPE increases the complexity of the extracted code by generating additional conditional logic and control operation semantics. Also, lambda expression–based refactoring[60] can be used to parameterize statements that could not be matched with any statement from the other clone instances because of an incompatible AST structure, but functional programming only supports for a Java 8–written program.[61] PRI classifies 4.1% of unrefactored clones as a variation of types or method calls. These incomplete refactorings occur because of limited language support for generic types or lack of a common code. PRI classifies 7 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 (eg, method overriding), but we do not observe such cases in our data sets.

REFFINDER[66] could be used to search for refactorings. While REFFINDER supports over 60 refactoring types from Fowler's catalog, several previous studies[82,83] point out the low precision and recall, including false-negative refactoring cases such as clones that were factored out into nested method calls. In contrast to the analysis of only VCS data in REFFINDER, PRI analyzes both clone groups in the clone database and source code in VCS to capture more precise data for identifying clone refactorings. We define clone refactoring pattern rules using the change pattern descriptions that represent the structural constraints before and after applying a refactoring to a clone group. In addition to using the skeleton of refactoring edits expressed as the logical constraints of program structural changes, PRI also leverages the clone detector outputs and focuses on clone evolutions based on 5 refactoring types that are the most commonly performed in practice, which we believe can achieve a higher accuracy of PRI's refactoring summarization and anomaly detection for clones.

5.3 | User study with computer science students

We recruited 8 participants from the Department of Computer Science at the University of Nebraska at Omaha. The participants included graduate students who have focused on the software engineering research area (software engineering concentration—https://goo.gl/zKUXt8). These students have taken several software engineering–related courses, including CSCI 8790 (Advanced Topics in Software Engineering), CSCI 8760 (Formal Method in Software Engineering), CSCI 8700 (Software Specification and Design), and CSCI 8986 (Advanced Tools for Software Development). The participant names are anonymized. These courses teach the principles of modern software engineering, including the tools, methods, and techniques that support their applications to offer a master's level project and dissertation.

All 8 participants had at least 4 years of Java development in academia, conducting team-based term projects within a semester time frame. They reported that refactorings and code reviews are regular activities during the development life cycle in their projects. Table 5 shows the demographic information on the 8 participants.

We introduced to the participants a study procedure that included a presentation of PRI's features and a 50-minute software demo of how to use the PRI plug-in built on top of Eclipse IDE. To get accurate and comprehensive feedback, participants were then asked to use PRI to investigate our motivating example and their term projects. We report that PRI is a mature tool that the participants did not have any issue in running on their codebase.

For each participant, the hand-on use of PRI lasted about 50 minutes, and then we conducted a semistructured interview to solicit their feedback on the utility of PRI. These interviews provide benefits of revealing unexpected types of information to be recorded.[84] The interview questions below were asked with an audio record for transcribing for further analysis.

- What kind of challenges do you face when you conduct refactorings to clones?
- In which situation do you think PRI can help improve refactorings to clones?
- Would you like to use PRI for your refactoring tasks to manage clones?
- How do you like or dislike PRI?

To provide insights on how refactorings are conducted and whether there are visible benefits of PRI, we present the interviews results organized by these questions during the interviews.

- *What kind of challenges do you face when you conduct refactorings?* Existing refactoring tools, such as Eclipse, can automatically perform and check the correctness of refactorings. However, participants find it hard to inspect refactorings to individual or a group of clone instances, since existing tools for refactorings require developers to explicitly invoke most refactoring features, lacking the ability to identify refactoring types applied to the same clone groups and detect overlooked mistakes.

  > *When conducting refactorings, I usually have a hard time finding the refactorable sets of code. Even if I find the code that is refactorable, it would take me so much more time to find all of the clone code that is in the same refactoring group. Therefore, the two biggest problems I have with refactoring are finding all of the code in the same refactorable group and the time it takes to do the refactoring.*
  > *When dealing with a large amount of code base for code reviews, refactoring candidates or mistakes do not easy to be found at the first place. It takes the time to check every portion of the program to make sure all the refactoring has been done in the correct way.*
  > *It is hard for us to quickly identify potential code fragments to be refactored and make decisions on whether the codes should be refactored.*

- *In which situation do you think PRI can help improve refactorings?* The participants mentioned that PRI can help them inspect system-wide refactorings to clones, so that they do not need to manually examine individual refactoring edits file by file. They also discussed that inspecting object-oriented programs usually requires analyzing dependence relationships, such as class inheritance hierarchies, to examine whether there are similar codes to be refactored in subclasses (eg, Pull Up Method). They believed that PRI's summary feature for clone refactorings is an efficient method for novices to inspect clone refactorings, unleashing the burdens of developers unaware of complex dependence relationships and spreading refactoring knowledge in a team.

  > *This tool can help me find all of the refactoring code and its location which will save me a lot of time looking through the code. With this tool, I do not have to worry about missing some of the same refactorable code. For example, assume that there is a refactorable*

TABLE 5 The demographic information of each study participant

Subject	Gender	Age	Java Experience (5)	Refactoring Proficiency (5)
1	Male	21-25	4	3
2	Female	25-30	3	2
3	Female	21-25	2	1
4	Female	21-25	3	5
5	Male	21-25	3	2
6	Male	21-25	3	2
7	Male	21-25	3	1
8	Female	25-30	4	2
9	Male	21-25	4	2
10	Female	25-30	2	4

clones in subclasses A, B, and C, but I only did the refactoring for Classes A and B since did not know about the refactorable code in class C. So with this tool, I think it can help save me a lot of time and complete my refactoring correctly without missing part.

PRI can improve refactoring by eliminating the probabilities of uncompleted clone refactoring. Also, other programmers who are working on the same program will understand what refactoring works have been done in the view that PRI reported.

When we do a large project with many team members working together, this clone refactoring tool is useful, because PRI's summary of clone groups saves the long time to read and understand other's code.

- *Would you like to use PRI for your refactoring tasks?* All 10 participants provided strong positive answers and believe that it would be helpful to use PRI for their refactoring tasks to manage clones.

  Yes, definitely. I would like to use this tool for my refactoring tasks. This tool makes job done in much easier way, also because it is easy to use.

  Yes, I would like to use this tool for code review. I would be more likely to use pull up method refactoring, and extract method refactoring if there are substantial code clones.

- *How do you like or dislike PRI?* Participants thought PRI can be used as a time-saving tool to inspect clone refactorings. Six participants replied that they like the search feature to identify potential refactorable clones in a large, evolving system because of its flexibility and interactivity compared with existing refactoring tools. Four participants mentioned that the user interface seems to be not intuitive for a first-time user, and they spent some time familiarizing with the user interface.

  I think the tool gives us a quicker way to identify potential code clone refactoring targets, which can save a lot of human intensive works that usually require manual code inspection. I would like to see more refactoring types that are beyond code clones.

  This tool can help me to reduce the time for checking each refactoring section of the program. It helps me to understand which portion of code has been refactored. Also, it can be a great help for me when cleaning up my code for easier future editing.

In summary, after using PRI to examine clone groups and refactoring changes, participants concluded that PRI can improve developer productivity in inspecting clone refactorings and they would like to use this tool in future projects. Student developers encounter challenges when conducting research projects that usually require system-wide refactorings to clones. Currently, in existing IDEs, they hardly use any reliable mechanisms to explore if all clones are correctly or consistently refactored. Participants think PRI would help them detect unobserved locations because it provides corrective messages for anomalies. The search feature for clone refactorings also makes it useful for student developers to use the tool. All participants strongly believe that they would like to use PRI to investigate the correctness of clone refactoring and detect potential refactoring errors, as a part of the refactoring task, knowing developers' intent to refactor.

6 | THREATS TO VALIDITY

Regarding studies on clone refactoring detection, in terms of *construct validity*, the accuracy of the clone detector[16] directly affects PRI's capability in clone refactoring summarization and refactoring anomaly detection. For example, as discussed in the *False Positives* section under Section 5.2, PRI produces false positives in that several groups are incorrectly classified since the clone detector suffers the inaccuracy in finding semantically equivalent behaviors between clones.

Since PRI is designed by a static analysis, it is currently unable to identify runtime object types precisely and to detect execution changes in the exception handling logic precisely.

PRI takes as input the original and refactored version snapshots; it does not use the data set that includes refactoring histories of which refactorings were executed (eg, a sequence of edit operations recorded in Eclipse), when they were performed, and with what configuration parameters during programming session. Improving PRI to analyze these refactoring data set[23,85,86] remains a future work.

We determine a threshold *Sim* in the clone refactoring detection algorithm in Section 4.1 by empirically measuring the success of the clone refactoring summarization for different threshold levels. Although we use the threshold *Sim* to evaluate particular refactoring pattern templates such as the Extract Method and Move Method in Table 1, the threshold *Sim* may vary for other pattern templates, such as the Pull Up Method or Extract Super Class, which typically does not change the method body significantly after a method is moved from a subclass. Exploring how the threshold affects each pattern templates can help future tool enhancements.

In terms of *internal validity*, PRI detects incomplete clone refactorings as potential refactoring anomalies. A developer might intend to independently evolve some clone instances rather than factoring out all clones. Nevertheless, we believe that paying attention to these locations and classifying whether they can easily be removed using standard refactoring techniques can be worthwhile during refactoring and maintenance tasks.

In terms of *external validity*, our results do not generalize beyond our data set and the subject applications. 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.

We investigated real developer's changes to use them as refactoring anomalies in the ground truth data set. These anomalies do not cover all possible refactoring-related bugs. However, these faults were collected by the authors' experience when performing and inspecting refactorings, and they were examined based on previous studies that identified issues in automated refactoring tools.

Refactoring of cloned code fragments has been observed in the evolution of many software systems. During the refactoring activities, developers remove the repetitive code by creating extract methods (ie, the Extract Method refactoring) or by moving generic methods in the superclass (ie, the Pull Up Method refactoring). After such modifications, developers describe refactoring of clones in commit logs of version control systems. In our case study, we have used the keyword matching heuristics to identify source code commits with clone refactoring, which is similar to other research of clone evolution.[87-89] However, during the manual inspection activities, we have observed that this approach can miss the majority of clone refactorings in our data sets since developers often fail to provide precise and complete descriptions in commit logs.[23] Therefore, we create 98 clone groups considered as moderate-sized data sets for our evaluation. Nevertheless, our future work collects more clone data sets by including different clone detection tools with more open-source projects.

Regarding the user study, we discuss the following threats to validity. In terms of *construct validity*, we created multiple-choice questions, such as usefulness and satisfaction when summarizing clone refactorings and detecting anomalies. Objective measures, such as time spent for refactoring inspections on clones, could be used to measure the same empirical matrices. However, our goal was to study the practical impact that the summarization and detection features for each refactoring types have on participants' opinions.

In terms of *internal validity*, there is a possible social desirability. Participants' perception could have been more positive toward PRI because they knew that the authors designed and implemented it. This threat, in fact, is hard to avoid, but we mitigated positive bias by indicating that PRI could execute inaccurately, reporting false positives or negatives, before starting the study. We observed that some participants checked PRI's results multiple times within a study session to decide whether to trust its effectiveness.

In terms of *external validity*, the first threat is the limited number of studied refactorings as well as recruited participants. By solely studying several refactoring types, we cannot generalize our study results to conclude that PRI can support equally well when helping developers manage other kinds of clone refactorings. Similarly, the count of recruited participants is trivial compared with the number of developers in the real world. We cannot conclude that PRI can help all developers achieve more accurate inspections during clone refactorings.

As a second threat, one may argue that our study's conclusions may not generalize to developers who have familiarity with their codebase. However, we do not believe that this concern is a significant limitation. Despite the possibility of their unfamiliarity with the study's code detail, we believe participants were familiar with the general aspects of a Java application. The clone refactorings applied, the anomalies, and their impacts were similar to common scenarios during activities for refactoring and maintenance.

7 | CONCLUSION

In this article, 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 uses refactoring pattern templates and traces cloned code fragments across revisions. It further analyzes refactoring anomalies to classify if developers cannot easily remove them using standard refactoring techniques. It also provides tool support for visualizing refactoring edit histories and for fixing refactoring 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 6 open-source projects. Our study with 10 computer science students shows that PRI is effective and helps them more efficiently trace and inspect clone refactorings in an evolving system. To the best of our knowledge, we are the first to present a technique that summarizes clone refactorings, detects refactoring anomalies, visualizes the clone refactoring evolution, and provides tool support to fix the anomalies. Its capability can help developers focus their attention on refactorings to clone groups across versions of a program and incomplete clone refactorings for a safe evolution.

As future work for improving PRI's approach and tool, we intend to (1) create pattern templates for more refactoring types, (2) provide tool support for more refactoring types for fixing anomalies, and (3) implement checking operations to determine the correctness of extra edits to pure refactoring versions by extending a prior work.[30] We also plan to conduct more user studies with professional developers to improve PRI's usability in industry-scale settings.

8 | AVAILABILITY

All experimental materials and collected data, including a software demo, are available online (https://sites.google.com/unomaha.edu/clonerefactoring).

ORCID

Myoungkyu Song http://orcid.org/0000-0003-4477-8933

REFERENCES

1. Hindle A, Barr ET, Su Z, Gabel M, Devanbu P. On the naturalness of software. In: Proceedings of the 34th International Conference on Software Engineering, *ICSE'12*. IEEE Press; 2012; Zurich, Switzerland:837-847.
2. Nguyen HA, Nguyen AT, Nguyen TT, Nguyen TN, Rajan H. A study of repetitiveness of code changes in software evolution. In: Proc. of ASE; 2013; Silicon Valley, CA, USA:180-190.
3. Roy CK, Cordy JR. An empirical study of function clones in open source software. In: 2008 15th Working Conference on Reverse Engineering. IEEE; 2008; Antwerp, Belgium:81-90.
4. Sasaki Y, Yamamoto T, Hayase Y, Inoue K. Finding file clones in FreeBSD ports collection. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE; 2010; Cape Town, South Africa:102-105.
5. Opdyke WF, Johnson RE. Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In: Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA). Poughkeepsie, NY; 1990.
6. Fowler M. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Professional; 2000.
7. Gode N, Harder J. Clone stability. In: Proc. of CSMR. IEEE; 2011; Oldenburg, Germany:65-74.
8. Krinke J. Is cloned code more stable than non-cloned code? In: Proc. of SCAM. IEEE; 2008; Beijing, China:57-66.
9. Krinke J. Is cloned code older than non-cloned code? In: Proc. of IWSC. ACM; 2011; New York, NY, USA:28-33.
10. Kim M, Bergman L, Lau T, Notkin D. An ethnographic study of copy and paste programming practices in OOPL. In: ISESE'04: Proceedings of the 2004 International Symposium on Empirical Software Engineering. IEEE Computer Society; 2004; Washington, DC, USA:83-92.
11. Barbour L, Khomh F, Zou Y. Late propagation in software clones. In: Proc. of ICSM. IEEE; 2011; Williamsburg, VI, USA:273-282.
12. Jiang L, Su Z, Chiu E. Context-based detection of clone-related bugs. In: Proc. of ESEC-FSE. ACM; 2007; New York, NY, USA:55-64.
13. Mondal M, Roy CK, Rahman MS, Saha RK, Krinke J, Schneider KA. Comparative stability of cloned and non-cloned code: an empirical study. In: Proc. of SAC. ACM; 2012; New York, NY, USA:1227-1234.
14. Roy CK, Cordy JR. Nicad: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: 2008. ICPC 2008. The 16th IEEE International Conference on Program Comprehension. IEEE; 2008; Amsterdam, The Netherlands, The Netherlands:172-181.
15. Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans Softw Eng*. 2002;28:654-670.
16. Jiang L, Misherghi G, Su Z, Glondu S. Deckard: scalable and accurate tree-based detection of code clones. In: Proc. of ICSE. IEEE; 2007; Washington, DC, USA:96-105.
17. Gabel M, Jiang L, Su Z. Scalable detection of semantic clones. In: Proc. ICSE; 2008; Leipzig, Germany:321-330.
18. Higo Y, Kusumoto S. Identifying clone removal opportunities based on co-evolution analysis. In: Proceedings of the 2013 International Workshop on Principles of Software Evolution. ACM; 2013; New York, NY, USA:63-67.
19. Higo Y, Kusumoto S, Inoue K. A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *J Softw Maintenance Evol Res Pract*. 2008;20(6):435-461.
20. Roy CK, Cordy JR, Koschke R. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *Sci Comput Prog*. 2009;74(7):470-495.
21. Kim M, Zimmermann T, Nagappan N. A field study of refactoring challenges and benefits. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM; 2012; New York, NY, USA:50.
22. Murphy GC, Kersten M, Findlater L. How are Java software developers using the Eclipse IDE *IEEE Softw*. 2006;23(4):76-83.
23. Murphy-Hill E, Parnin C, Black AP. How we refactor, and how we know it. *IEEE Trans Softw Eng*. 2012;38(1):5-18.
24. Negara S, Chen N, Vakilian M, Johnson RE, Dig D. A comparative study of manual and automated refactorings. In: European Conference on Object-Oriented Programming; 2013; Berlin, Heidelberg:552-576.
25. Vakilian M, Chen N, Negara S, Rajkumar BA, Bailey BP, Johnson RE. Use, disuse, and misuse of automated refactorings. In: Proc. of ICSE; 2012; Zurich, Switzerland:233-243.
26. Rachatasumrit N, Kim M. An empirical investigation into the impact of refactoring on regression testing. In: ICSM'12: the 28th IEEE International Conference on Software Maintenance. IEEE Society; 2012; Trent, Italy:10.
27. Weißgerber P, Diehl S. Are refactorings less error-prone than other changes? In: MSR'06: Proceedings of the 2006 International Workshop on Mining Software Repositories. ACM; 2006; New York, NY, USA:112-118.
28. Dig D, Johnson R. The role of refactorings in API evolution. In: ICSM'05: Proceedings of the 21st IEEE International Conference on Software Maintenance. IEEE Computer Society; 2005; Washington, DC, USA:389-398.
29. Plug-in development environment: http://www.eclipse.org/pde.
30. Alves EL, Song M, Kim M. RefDistiller: a refactoring aware code review tool for inspecting manual refactoring edits. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM; 2014; New York, NY, USA:751-754.
31. Ge X, Sarkar S, Murphy-Hill E. Towards refactoring-aware code review. In: Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering. ACM; 2014; New York, NY, USA:99-102.
32. Ge X, Sarkar S, Witschey J, Murphy-Hill E. Refactoring-aware code review. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'17); 2017; Raleigh, NC, USA:71-79.
33. Li Z, Lu S, Myagmar S, Zhou Y. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In: Proc. of OSDI; 2004; San Francisco, CA:289-302.

34. Pham NH, Nguyen TT, Nguyen HA, Nguyen TN. Detection of recurring software vulnerabilities. In: Proc. of ASE. ACM; 2010; New York, NY, USA: 447-456.
35. Li J, Ernst MD. CBCD: cloned buggy code detector. In: Proc. of ICSE. IEEE; 2012; Zurich, Switzerland:310-320.
36. Ray B, Kim M, Person S, Rungta N. Detecting and characterizing semantic inconsistencies in ported code. In: Proc. of ASE. IEEE; 2013; Silicon Valley, CA, USA:367-377.
37. Göde N, Koschke R. Studying clone evolution using incremental clone detection. *J Softw Evol Process*. 2013;25(2):165-192.
38. Saha RK, Roy CK, Schneider KA, Perry DE. Understanding the evolution of type-3 clones: an exploratory study. In: Proc. of MSR. IEEE; 2013; San Francisco, CA, USA:139-148.
39. Bettenburg N, Shang W, Ibrahim W, Adams B, Zou Y, Hassan AE. An empirical study on inconsistent changes to code clones at release level. In: Proc. of WCRE. IEEE; 2009; Lille, France:85-94.
40. Krinke J. *A Study of Consistent and Inconsistent Changes to Code Clones*, *Proc. of WCRE*. Vancouver, BC, Canada: IEEE; 2007;170-178.
41. Aversano L, Cerulo L, Penta MD. How clones are maintained: an empirical study. In: Proc. of CSMR. IEEE; 2007; Washington, DC, USA:81-90.
42. Toomim M, Begel A, Graham SL. Managing duplicated code with linked editing. In: Proc. VLHCC. IEEE; 2004; Rome, Italy:173-180.
43. Duala-Ekoko E, Robillard MP. Tracking code clones in evolving software. In: Proc. of ICSE. IEEE; 2007; Minneapolis, MN, USA:158-167.
44. Hou D, Jablonski P, Jacob F. CnP: towards an environment for the proactive management of copy-and-paste programming. In: Proc. ICPC. IEEE; 2009; Vancouver, BC, Canada:238-242.
45. Nguyen TT, Nguyen HA, Pham NH, Al-Kofahi JM, Nguyen TN. Clone-aware configuration management. In: Proc of ASE; 2009; Auckland, New Zealand:123-134.
46. Jablonski P, Hou D. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE. In: Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology eXchange. ACM; 2007; New York, NY, USA:16-20.
47. Lin Y, Xing Z, Xue Y, Liu Y, Peng X, Sun J, Zhao W. Detecting differences across multiple instances of code clones. In: Proc. of ICSE. ACM; 2014; New York, NY, USA:164-174.
48. Hou D, Jacob F, Jablonski P. Exploring the design space of proactive tool support for copy-and-paste programming. In: Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research. IBM Corp; 2009; Riverton, NJ, USA:188-202.
49. Jacob F, Hou D, Jablonski P. Actively comparing clones inside the code editor. In: Proceedings of the 4th International Workshop on Software Clones. ACM; 2010; New York, NY, USA:9-16.
50. Tsantalis N, Chatzigeorgiou A. Identification of extract method refactoring opportunities for the decomposition of methods. *J Syst Softw*. 2011;84(10):1757-1782.
51. Bavota G, De Lucia A, Marcus A, Oliveto R, Palomba F. Supporting extract class refactoring in Eclipse: the ARIES project. In: ProcC of ICSE. IEEE; 2012; Zurich, Switzerland:1419-1422.
52. Ge X, DuBose QL, Murphy-Hill E. Reconciling manual and automatic refactoring. In: Proc. of ICSE. IEEE; 2012; Zurich, Switzerland:211-221.
53. Foster SR, Griswold WG, Lerner S. Witchdoctor: IDE support for real-time auto-completion of refactorings. In: Proc. of ICSE; 2012; Zurich, Switzerland:222-232.
54. Kim M, Sazawal V, Notkin D, Murphy G. An empirical study of code clone genealogies. In: Proc. of ESEC/FSE; 2005; New York, NY, USA:187-196.
55. Mondal M, Roy CK, Schneider KA. Automatic ranking of clones for refactoring through mining association rules. In: 2014 Software Evolution Week—IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE). IEEE; 2014; Antwerp, Belgium:114-123.
56. Tsantalis N, Mazinanian D, Krishnan GP. Assessing the refactorability of software clones. *IEEE Trans Softw Eng*. 2015;41(11):1055-1090.
57. Tairas R, Gray J. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Inf Softw Technol*. 2012;54(12):1297-1307.
58. Hotta K, Higo Y, Kusumoto S. Identifying, tailoring, and suggesting form template method refactoring opportunities with program dependence graph. In: 2012 16th European Conference on Software Maintenance and Reengineering (CSMR). IEEE; 2012; Szeged, Hungary:53-62.
59. Bian Y, Koru G, Su X, Ma P. SPAPE: a semantic-preserving amorphous procedure extraction method for near-miss clones. *J Syst Softw*. 2013;86(8): 2077-2093.
60. Tsantalis N, Mazinanian D, Rostami S. Clone refactoring with lambda expressions. In: Proceedings of the 39th International Conference on Software Engineering. IEEE Press; 2017; Buenos Aires, Argentina:60-70.
61. Urma RG, Fusco M, Mycroft A. *Java 8 in Action: Lambdas, Streams, and Functional-Style Programming*. Shelter Island, NY: Manning Publications Co.; 2014.
62. Barnett M, Bird C, Brunet J, Lahiri S. Helping developers help themselves: automatic decomposition of code review changesets. In: Proc. of ICSE. IEEE; 2015; Piscataway, NJ, USA:134-144.
63. Tao Y, Dang Y, Xie T, Zhang D, Kim S. How do software engineers understand code changes? An exploratory study in industry. In: Proc. of FSE. ACM; 2012; New York, NY, USA:51.
64. Fluri B, Würsch M, Pinzger M, Gall HC. Change distilling—tree differencing for fine-grained source code change extraction. *IEEE Trans Softw Eng*. 2007;33(11):725-743.
65. Tufano M, Palomba F, Bavota G, et al.. There and back again: can you compile that snapshot? *J Softw Evol Process*. 2017;29(4).
66. Prete K, Rachatasumrit N, Sudan N, Kim M. Template-based reconstruction of complex refactorings. In: Proc. of ICSM; 2010; Timişoara, Romania:1-10.
67. Levenstein VI. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10*. 1966;10(8):707-710.
68. Meng N, Hua L, Kim M, McKinley KS. Does automated refactoring obviate systematic editing? In: Proc. of ICSE. IEEE; 2015; Florence, Italy:392-402.
69. Chesley OC, Ren X, Ryder BG. Crisp: a debugging tool for Java programs. In: Proc. ICSM. IEEE; 2005; Budapest, Hungary, Hungary:401-410.
70. Zhang T, Song M, Pinedo J, Kim M. Interactive code review for systematic changes. In: Proc. of ICSE; 2015; Florence, Italy:111-122.
71. Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. *J TPLS*. 1990;12(1):26-60.

72. Mongiovi M, Mendes G, Gheyi R, Soares G, Ribeiro M. Scaling testing of refactoring engines. In: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE; 2014; Victoria, BC, Canada:371-380.

73. Soares G, Gheyi R, Massoni T. Automated behavioral testing of refactoring engines. *IEEE Trans Softw Eng*. 2013;39(2):147-162.

74. Soares G, Mongiovi M, Gheyi R. Identifying overly strong conditions in refactoring implementations. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM). IEEE; 2011; Williamsburg, VI, USA:173-182.

75. Vakilian M, Johnson RE. Alternate refactoring paths reveal usability problems. In: Proceedings of the 36th International Conference on Software Engineering. ACM; 2014; New York, NY, USA:1106-1116.

76. Murphy-Hill E, Black AP. Breaking the barriers to successful refactoring: observations and tools for extract method. In: ICSE'08: Proceedings of the 30th International Conference on Software Engineering. ACM; 2008; New York, NY, USA:421-430.

77. Daniel B, Dig D, Garcia K, Marinov D. Automated testing of refactoring engines. In: ESEC-FSE'07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM; 2007; New York, NY, USA:185-194.

78. Mongiovi M, Gheyi R, Soares G, Teixeira L, Borba P. Making refactoring safer through impact analysis. *Sci Comput Prog*. 2014;93:39-64.

79. Manning CD, Surdeanu M, Bauer J, Finkel JR, Bethard S, McClosky D. The Stanford CoreNLP natural language processing toolkit. In: ACL (System Demonstrations); 2014; Baltimore, MD:55-60.

80. Krinke J. Identifying similar code with program dependence graphs. In: Proc. of WCRE. IEEE; 2001; Stuttgart, Germany:301-309.

81. Miller RC, Myers BA. Interactive simultaneous editing of multiple text regions. In: Proceedings of the General Track: 2002 USENIX Annual Technical Conference. USENIX Association; 2001; Berkeley, CA, USA:161-174.

82. Kádár I, Hegedús P, Ferenc R, Gyimóthy T. A manually validated code refactoring dataset and its assessment regarding software maintainability. In: Proceedings of the 12th International Conference on Predictive Models and Data Analytics in Software Engineering. ACM; 2016; New York, NY, USA:10.

83. Soares G, Gheyi R, Murphy-Hill E, Johnson B. Comparing approaches to analyze refactoring activity on software repositories. *J Syst Softw*. 2013;86(4):1006-1022.

84. Seaman CB. Qualitative methods in empirical studies of software engineering. *IEEE Trans Softw Eng*. 1999;25(4):557-572.

85. Negara S, Vakilian M, Chen N, Johnson RE, Dig D. Is it dangerous to use version control histories to study source code evolution?*ECOOP*. 2012;12:79-103.

86. Vakilian M, Chen N, Negara S, Rajkumar BA, Zilouchian Moghaddam R, Johnson RE. The need for richer refactoring usage data. In: Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, *PLATEAU'11*. ACM; 2011; New York, NY, USA:31-38.

87. Göde N. Clone removal: fact or fiction? In: Proceedings of the 4th International Workshop on Software Clones. ACM; 2010; New York, NY, USA:33-40.

88. Parnin C, Görg C. Improving change descriptions with change contexts. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories. ACM; 2008; New York, NY, USA:51-60.

89. Wang W, Godfrey MW. Investigating intentional clone refactoring. *Electron Commun EASST*. 2014:63.

How to cite this article: Chen Z, Kwon Y-W, Song M. Clone refactoring inspection by summarizing clone refactorings and detecting inconsistent changes during software evolution. *J Softw Evol Proc*. 2018;30:e1951. https://doi.org/10.1002/smr.1951