This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2679742, IEEE Transactions on Software Engineering

Refactoring Inspection Support for Manual Refactoring Edits

Everton L. G. Alves, Myoungkyu Song, Tiago Massoni, Patrícia D. L. Machado, Miryung Kim

Abstract—Refactoring is commonly performed manually, supported by regression testing, which serves as a safety net to provide confidence on the edits performed. However, inadequate test suites may prevent developers from initiating or performing refactorings. We propose REFDISTILLER, a static analysis approach to support the inspection of manual refactorings. It combines two techniques. First, it applies predefined templates to identify potential missed edits during manual refactoring. Second, it leverages an automated refactoring engine to identify extra edits that might be incorrect. REFDISTILLER also helps determine the root cause of detected anomalies. In our evaluation, REFDISTILLER identifies 97% of seeded anomalies, of which 24% are not detected by generated test suites. Compared to running existing regression test suites, it detects 22 times more anomalies, with 94% precision on average. In a study with 15 professional developers, the participants inspected problematic refactorings with REFDISTILLER vs. testing only. With REFDISTILLER, participants located 90% of the seeded anomalies, while they located only 13% with testing. The results show REFDISTILLER can help check the correctness of manual refactorings.

Index Terms—Refactoring, Refactoring Anomalies, Code Inspection

1 INTRODUCTION

Developers often refactor systems manually [29]. According to recent studies, developers do not use refactoring tools, despite their awareness of automated refactoring in Integrated Development Environments (IDE) [25], [32], [46]. Murphy et al. find that about 90% of refactoring edits are done manually [29]. Negara et al. show that expert developers prefer manual to automated refactoring [32]. Vakilian et al. find that automated refactoring is underused and misused [46].

Despite the original intention of improving software quality and productivity through refactoring, manual application of refactoring edits often leads to error-proneness. Several studies find there is a strong correlation between the timing and location of refactorings and bug fixes [24].[1] Weißgerber and Diehl find that a high ratio of refactoring is often followed by an increasing ratio of bug reports [24], [47]. Bavota et al. find that refactorings involving hierarchies induce faults frequently [8]. In a field study of refactoring at Microsoft, 77% of the survey participants perceive that refactoring comes with a risk of introducing subtle bugs and functionality regression. Developers find it difficult to ensure the correctness of manual refactoring—*"I would like code understanding and visualization tools to help me make sure that my manual refactorings are valid."* (a quote from a professional developer [25]).

Existing approaches for ensuring correctness of manual refactoring are limited. Rachatasumrit and Kim find that regression testing suites in practice are often inadequate for covering refactored locations and ineffective in detecting refactoring errors [39]. SafeRefactor validates refactoring edits by leveraging an existing test generation engine and by comparing test results between the old and new program versions [43]. However, it also requires having enough test coverage. GhostFactor validates whether certain behavior preserving conditions are satisfied for refactoring edits in C# [20]; however, it detects only missing edits and does not detect extra edits that may change a program's behavior.

Due to their complexity and error-proneness, refactoring edits require special attention during maintenance and inspection tasks [1], [2]. Ge et al.'s survey with professional developers lists four reasons why refactoring inspection deserves a special attention [48]: (1) refactoring helps to transmit design decisions and conventions; (2) the motivation for refactoring needs to be justified; (3) refactoring exposes previously hidden problems, and (4) a refactoring patch needs to be validated whether non-refactoring changes are intermingled or not.

This article proposes REFDISTILLER [3], a static analysis approach and tool to help developers inspect refactorings. REFDISTILLER detects possible refactoring anomalies and provides useful information regarding the location of the detected refactoring anomalies and their possible causes. The tool focuses on two classes of refactoring anomalies: *missing* and *extra edits*. In the first phase, **M**issing **E**dit Anomaly **Detector** (MEDETECTOR) searches for constituent edits that may have been neglected by the developer. In the second phase, **E**xtra **E**dit Anomaly **Detector** (EEDETECTOR) searches for extra edits that are beyond the required refactoring steps. REFDISTILLER currently supports six widely used refactoring type [31]: extract method, inline method, move method, pull up method, push down method, and rename method.

REFDISTILLER reports the type of a refactoring anomaly, its location, and additional hints about how to fix the anomaly. Since REFDISTILLER provides a pure refactoring version for comparison, the pure refactoring version can be used as a starting point to fix the detected anomaly.

We evaluate REFDISTILLER using three different meth-

1. In this article, we use the terms bug, fault, anomaly and refactoring problem interchangeably. In our study, participants also used these terms when discussing refactoring issues.

ods. First, we apply it to a data set composed by 100 refactoring transformations with seeded refactoring anomalies. These refactoring errors do not generate any compilation errors, and many of them are not checked by state-of-the-art automated refactoring engines such as Eclipse. 96% of the anomalies were detected by RefDistiller, and 24 of these detected anomalies were missed by the generated test suites. We also apply RefDistiller to three open source projects, XMLSecurity, JMeter, and JMock, to assess its performance on real-world scenarios. RefDistiller presented a 94% precision, whereas 97% of anomalies are not found by the existing test suite for each project written by developers. Finally, we conduct a user study with 15 professional developers, in which they found RefDistiller useful for inspecting refactoring, having saved them an average of 17% inspection time. The participants also were able to detect and locate more refactoring anomalies using RefDistiller than using tests alone.

In summary, our paper makes the following contributions:

- We present RefDistiller, a novel static analysis approach and an open source implementation to detect anomalies in manual refactoring.[2] Its scope is currently the most comprehensive among existing techniques, covering six widely used refactoring types and it focuses on anomalies that do not generate any compilation errors.
- By providing debugging information regarding the location and potential cause of refactoring anomalies, RefDistiller helps developers to double check whether their manual refactoring edits are correct.
- Our empirical evaluation shows that RefDistiller can detect potential missing and extra edits with high precision. Our static analysis approach can effectively complement testing by detecting more anomalies.
- Our user study shows that RefDistiller can help detect more anomalies and save inspection time by providing useful feedback, compared to running tests only.

2 Motivating Example

This section motivates our approach and illustrates how RefDistiller would help a developer in inspecting refactoring edits.

Suppose Bob works as a developer in the XMLSecurity project, a library that provides security APIs for managing XML files. While working on several tasks, Bob notices an opportunity to reduce code duplication and decides to perform an extract method refactoring manually. Figures 1(a) and (b) show the edit performed by Bob: code insertion is marked with '+', deletion with '−'. Bob extracts line 4 to a new method `initializeDigest` and adds a call at line 5. However, although aiming a behavior-preserving edit, he forgets to update variable `digestValueElement` with the return value of `initializeDigest()` (a newly created method)–line 5. Bob does not suspect that he mistakenly

2. https://sites.google.com/site/refdistiller/

```
1  class Reference {
2    boolean verify() throws .. {
3      Element digestValueElem = (Element) new Node(0);
4      digestValueElem = this.getChildElementLocalName(..);
5      byte[] p1 = Base64.decode(digestValueElem);
6      byte[] p2 = this.calculateDigest();
7      boolean re = MessageDigestAlgorithm.isEqual(p1, p2);
8      if (!re) { .. }
9      return re;
10   }
11   Element updateElement(){ ... }
12 }
13
14 class XMLCipher {
15   void addTransform(Transform transform) {
16     transforms.add(transform);
17   }
18 }
19
20 class Transformer { .. }
```

(a) original code

```
1  class Reference {
2    boolean verify() throws .. {
3      Element digestValueElem = (Element) new Node(0);
4  -   digestValueElem = this.getChildElementLocalName(..);
5  +   initializeDigest();// This line should be
       digestValueElem = initializeDigest();
6      byte[] p1 = Base64.decode(digestValueElem);
7      byte[] p2 = this.calculateDigest();
8      boolean re = MessageDigestAlgorithm.isEqual(p1, p2);
9      if (!re) { .. }
10     return re;
11   }
12
13 + Element initializeDigest() {
14 +   Element digestValueElem;
15 +   digestValueElem = this.getChildElementLocalName(..);
16 +   return digestValueElem;
17 + }
18 Element updateElement(){ ... }
19 }
```

(b) Extract Method refactoring with missing edits

```
1  class XMLCipher {
2  - void addTransform(Transform transform) {
3  -   transforms.add(transform);
4  - }
5  }
6
7  class Transformer {
8  + void addTransform(Transform transform) {
9  +   try {
10 +     Transform transform = Transform.getInstance(..);
11 +     transforms.add(transform);
12 +   } catch (InvalidTransformException ex) {
13 +     ..
14 +   }
15 + }
16 }
```

(c) Move Method refactoring with extra edits

Fig. 1. An example of problematic refactoring edits. (a) The original code. (b) Code after Bob's Extract Method refactoring. Lines 4 is extracted to create a new method `initializeDigest`. Variable `digestValueElem` should have been updated with the return value of the new method. Because there is no compilation error, Bob misses the required edit. (c) Code after Bob's Move Method refactoring. During the Move Method refactoring, underlined statements at lines 9–10 and 12–14 are added as extra edits in `addTransform`

changed the program semantics, since there is no compilation error and existing tests in XMLSecurity still pass. This is an example where a missing edit is erroneously introduced.

When using RefDistiller for inspecting his refactoring, Bob or a second inspector would easily identify the problem since the MEDetector module applies a static analysis to detect this missing edit and locates the potential root cause of the problem. It reports that there is a problem with the statement updated at line 5. It should

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2017.2679742, IEEE Transactions on Software Engineering

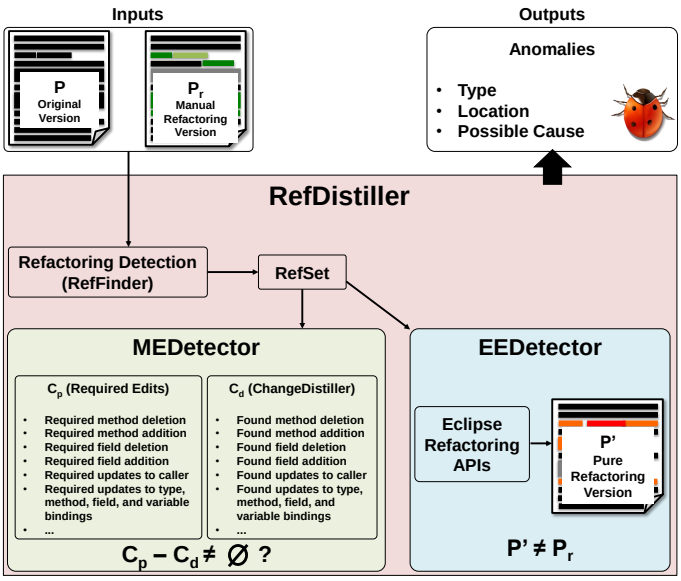

Fig. 2. RefDistiller overview.

be `digestValueElem = initializeDigest();` instead of `initializeDigest();`.

Bob performs a move method refactoring (Figures 1(a) and (c)). He moves method `addTransform` from class `XMLCipher` to class `Transformer`. During this refactoring, he adds new statements at lines 9–10 (underlined) and 12–14. Although this extra edit may be Bob's intended edit, it may be worthwhile for others to note this behavior change during code inspection or for Bob to reconfirm his intent, because this edit deviates from the pure move method edit. By comparing a pure-refactoring version against the manual refactoring version, EEDETECTOR locates and highlights the differences at lines 9–10 and 12–14.

The aforementioned missing and extra edits do not create any compilation errors. Thus, they are likely to pass unnoticed. The extra edits made during refactoring are not always errors and could be intentional; however, revisiting these extra edits during peer reviews can be useful [25].

3 REFDISTILLER

Figure 2 illustrates REFDISTILLER. It first takes as input the original version P and the manually refactored version P_r, applying RefFinder [38] to infer the types and locations of refactoring edits performed between P and P_r, which we call *RefSet*. With this information, REFDISTILLER detects potential refactoring anomalies. We define refactoring anomalies as edits that are not always bugs but often lead to behavioral changes. In the following sections, we discuss the two REFDISTILLER modules: MEDETECTOR, responsible for detecting missing edits and EEDETECTOR, responsible for revealing extra edits. REFDISTILLER is implemented as an Eclipse plug-in, and it is available to download at its website.[3]

3.1 MEDETECTOR: Detecting Missing Edits

MEDETECTOR applies static analysis for detecting edits that might have been missed by a developer during manual refactoring. For each refactoring type, we define template rules to describe required constituent edits (Table 1). The rules are based on Fowler's mechanics for correct refactorings [17] and the literature about formal specifications of refactorings [9], [17], [35], [40]. Moreover, the rules were also inspired by common refactoring anomalies reported in the literature [8] and the authors' experience with conducting refactorings in real projects. We target the following six refactoring types, as they are the most commonly applied [29]: extract method, inline method, move method, pull up method, push down method, and rename method. Each refactoring type requires a proper refactoring template. For each matched refactoring edit in *RefSet*, MEDETECTOR compares a set of expected edits C_p with actual manual edits C_d, generated by ChangeDistiller [14]. ChangeDistiller extracts source code changes using tree differencing, reporting change types such as adding or deleting a method, inserting or removing a statement, or changing the modifier of a method declaration.

For each refactoring type in RefSet, MEDETECTOR updates the set of required method and field reference updates, C_p, with the constituents edits based on predefined template rules. A new required edit is a triple `<Type, Element, Location>`, where `Location` represents the specific code element where the edit is expected (e.g., class, method, and field), `Element` represents the code element involved in the edit, and `Type` indicates the type of expected edit. For instance, `<'Update Statement',` $m2_n$, $m1_o$`>` defines the edit adding an assignment statement into $m1_o$, with $m2_n$ referenced in the right-hand side of the assignment.

When assessing edits, it is important to check whether method calls, field accesses, and type declarations are preserved before and after manual refactoring. Therefore, we create a new checking type `Binding Problem`, which is not currently supported by ChangeDistiller. MEDETECTOR includes a binding object to C_p every time a reference, e.g., a method call, a variable access, or a field access, differs in the original code, from its correspondent in the refactored code. Those references are syntactically associated with the respective class and object scope. Our binding-checking strategy compares AST trees in order to verify preservation of desired references. MEDETECTOR verifies whether the associations remain for modified methods, their callers, and their direct callees. Similarly, it checks all refactored fields and corresponding accessor methods.

MEDETECTOR reports the following nine types of warning messages for mismatches between C_p and C_d:

- *Binding problem:* Detected a problematic reference binding X which may have affected method Y.
- *Missing method:* Method X was not found in Class Y.
- *Missing statement update:* There is at least one missing statement to be updated in the method X's body.
- *Missing statement addition:* There is at least one missing statement to be inserted in the method X's body.
- *Missing statement deletion:* There is at least one missing statement to be deleted in the method X's body.

3. https://sites.google.com/site/refdistiller/

TABLE 1
The refactoring change rules of the MEDETECTOR's templates.

Extract Method (P: original version, P_r: modified version, $m1_o$: method to be refactored, $m2_n$: extracted method, $[startLine, endLine]$: portion to be extracted)		
1	C_p ={}; c_o = **getClass**(P, $m1_o$); C_p = C_p ∪ {<'**Add Functionality**', $m2_n$, c_o>}	Modified version P_r must include a method not existing in original version P.
2	STM_o = **getStatements** ($m1_o$, $[startLine, endLine]$) **IF** (**haveDependencies** (P, STM_o)) **THEN** C_p = C_p ∪ {<'**Update Statement**', $m2_n$, $m1_o$>};	If any extracted statements (STM_o) modify the value of variable(s) used in the rest of the method, the modified method must have a new variable update. The updated variable must be associated to the return value of a calling to the new method.
3	**IF** (**NOT haveDependencies** (P, STM_o)) **THEN** C_p = C_p ∪ {<'**Insert Statement**', $m2_n$, $m1_o$>};	If Rule 2 is not applicable, there must be a new statement related to the calling of the new method in the modified version.
4	**FOREACH** (s **in** STM_o) **DO** C_p = C_p ∪ {<'**Delete Statement**', s, $m1_o$>}; C_p = C_p ∪ {<'**Insert Statement**', s, $m2_n$>};	For each extracted statement, there must be a deleted statement in the original method $m1_o$ and an inserted statement (same statement) in the modified method $m2_n$. The order of inserted statements must be the same as extracted.
5	C = **getCallers**(P, $m1_o$); **FOREACH** (c **in** C) **DO** m = **getMethod**(P_r, c); $BindingProbs$ = **checkBindingProblem**(c, m); **FOREACH** (b **in** $BindingProbs$) **DO** C_p = C_p ∪ {<'**Binding Problem**', b, c>};	All callers of $m1_o$ in the modified version (m from P_r) must preserve all method and variable references from the original version P.
6	C = **getCallers**(P_r, $m2_n$); **FOREACH** (c **in** C) **DO** m = **getMethod**(P, c); $BindingProbs$ = **checkBindingProblem**(c, m); **FOREACH** (b **in** $BindingProbs$) **DO** C_p = C_p ∪ {<'**Binding Problem**', b, c>};	All callers of methods with similar signature (same name but different parameters) of $m2_n$ in the modified version must preserve all method and variable references from the original version P.
7	$unionSet$ = $m1_o$ ∪ $m2_n$; $BindingProbs$ = **checkBindingProblem**($m1_o$, $unionSet$); **FOREACH** (b **in** $BindingProbs$) **DO** C_p = C_p ∪ {<'**Binding Problem**', b, $m1_o$>};	The methods and variable references in $m1_o$ must be the combination of the references from the original method in the new version $m1_o$ and the newly added one $m2_n$, except the references affected by the application of Rule 2 or 3.
Inline Method (P: original code, P_r: modified version, $m1_o$: method to be inlined)		
8	C_p ={}; c_o = **getClass**(P, $m1_o$); C_p = C_p ∪ {<'**Remove Functionality**', c_o>}	There must be a deleted method in the modified version P_r.
9	STM_o = **getStatements** ($m1_o$, []); C = **getCallers**(P, $m1_o$); **FOREACH** (c **in** C) **DO** $m2_n$ = **getMethod**(P_r, **c**); **IF** (**isNotVoid** (c)) **THEN** C_p = C_p ∪ {<'**Update Statement**', c>}; **FOREACH** (s **in** STM_o) **DO** C_p = C_p ∪ {<'**Insert Statement**', **s**, c>};	If the inlined method $m1_o$ has a return type, there must be an updated statement in each of its callers. Also, for all callers, there must exist a sequence of inserted statement inlined from $m1_o$.
10	Same steps as in Rule 5.	See 5.
Move Method (P: original version, P_r: modified version, $m1_o$: method to be refactored, $m2_n$: newly added method)		
11	C_p = C_p ∪ {<'**Remove Functionality**', c_o>}	There must be a method deleted in the original version P.
12	c_o = **getClass**(P_r, $m1_o$); C_p = C_p ∪ {<'**Add Functionality**', c_o>}	There must be a new method in the modified version P_r.
13	C = **getCallers**(P, $m1_o$); **FOREACH** (c **in** C) **DO** **IF** (**isAssociatedWithField**($m1_o$)) **THEN** C_p = C_p ∪ {<'**Change Attribute Type**', c>}; **ELSE** C_p = C_p ∪ {<'**Update Statement**', c>};	For all callers of $m1_o$, if the method call is associated to a field, there must be an attribute type change in P and a statement update otherwise.
14	Same steps as in Rule 5.	See 5.
15	**IF** (**checkBindingProblem**($m1_o$, $m2_n$)) **THEN** C_p = C_p ∪ {<'**Binding Problem**', $m1_o$>};	All method and variable references in $m1_o$ must remain the same in the modified version P_r.
16	C = **getCallers**(P, $m1_o$); **FOREACH** (c **in** C) **DO** m = **getMethod**(P_r, c); **IF** (**verifyAccessibilityChange** (c, m)) **THEN** C_p = C_p ∪ {<'**Change Visibility**', m>};	Added method $m2_n$ must be visible to the callers of the removed method $m1_o$.
Pull Up Method: rules 5, 11, 12, and 15.		
Push Down Method: rules 5, 11, 12, and 15.		
Rename Method (P: original code, P_r: modified version, $m1_o$: method to be renamed, $m2_n$: renamed method)		
17	C_p ={}; c_o = **getClass**(P, $m2_n$); C_p = C_p ∪ {<'**Rename Method**', c_o>}	There must be a method in P_r that had its signature changed when compared to the original version P.
18	C = **getCallers**(P, $m2_n$); **FOREACH** (c **in** C) **DO** C_p = C_p ∪ {<'**Update Statement**', c>};	For all callers to $m2_n$, there must be an updated statement in modified version P_r.
19	C = **getCallers** (P, $m1_o$); C_r = **getCallers** (P_r, $m2_n$); **IF** ($C \neq C_r$) **THEN** C_p = C_p ∪ {<'**Binding Problem**', $m2_n$>};	The set of callers of the original method must be identical after the renaming.

- *Missing type update:* The type associated with field X needs to be updated.
- *Missing renaming:* Method X was not renamed.
- *Not removed method:* Method X should have been removed from Class Y.
- *Visibility problem:* Method X is not visible for one of its callers.

To illustrate application of the refactoring templates, we discuss the template rules for the extract method refactoring, in which the newly created method is placed in the same class from which the code was extracted. However, extraction to a different class can be decomposed into extracting the method first, then moving the method. Rules 1-7 in Table 1 define steps to check whether the extraction is successful. The rules focus on (i) correct placement of the newly created method and extracted statements, (ii) reference consistency with respect to its callers, and (iii) scoping consistency with respect to parameters and local variables. Failing to perform any of those steps may lead to behavioral changes. Rules 5, 6 and 7 perform a binding check to verify whether the methods and fields still reference the equivalent AST elements after the refactoring edits. This binding analysis is important, since simple code modifications may lead to subtle errors in variable and method references. For example, the newly created method could unexpectedly override a method in a superclass. Similarly, we check whether a renamed method overrides an inherited method in the refactored version. The rules are presented in pseudo code, and the following shows the meaning of the auxiliary functions:

- **`getClass(P, m)`** returns the containing class of method `m` to be refactored.
- **`getCallers(m)`** returns all callers of method `m`.
- **`getStatements(m, [beginLine;endLine])`** returns the statements from line `beginLine` to line `endLine`. If an empty range (i.e, []) is given, it returns all statements from method `m`.
- **`checkBindingProblem(m1, m2)`** verifies whether all references to methods or variables within `m1` and `m2` are identical, returning a set of problematic reference objects.
- **`haveDependencies(m, stm)`** verifies whether the remaining statements in method `m`, after extracting statements **`stm`** from method `m`, depend on any statement within extracted statements `stm`.
- **`isAssociatedWithAField(m)`** verifies whether a designated method `m` accesses any field declared in the same class.

The rules depicted in Table 1 are not final; due to the inherent complexity of manual refactorings in object-oriented systems, it is hard to glimpse all possible effects they might produce in a code base. These rules cover the most important constituent edits that, when missed (not checked), often lead to behavioral changes. However, the rule set may be extended in the future, when the users of REFDISTILLER report additional cases of common refactoring mistakes.

The rules in Table 1 cover some of the six most common refactoring types [31]. However, other refactoring edits may share similar constituent steps. For instance, similar to pull up method, the pull up field refactoring takes a field declared in a specific class and moves it to the superclass. Therefore, a pull up field template could be based on the pull up method template and a subset of the same rules can be reused. Thus, we believe REFDISTILLER can be easily extended to support other refactoring types.

TABLE 2
Conditions checked by EEDETECTOR. When any of these situations happen in the context of a Push Down Method, the Eclipse engine will generate a "not" safe refactoring. EEDETECTOR avoids those cases.

Bug	*C1: class under refactorings; C2: target class; m: method under refactorings*
320115	Method m is to be pushed down, and it directly calls a method that is invisible from the target class.
348278	Method m is to be pushed down, and it contains a method call using the keyword *this*.
356698/355322	Method m contains a super access to a method that is overridden/overloaded in class $C1$.
290618/355324	Method m contains a call to a method that is overridden/overloaded in the target class $C2$.
195003	Method m is to be pushed down, which contains a field access using the keyword *this*.
195004	Method m is to be pushed down, whose callee invokes another method by the origin class (e.g., *new ClassX().foo()*.)

3.2 EEDETECTOR: Revealing Extra Edits

Recent studies show that developers often apply refactoring in the context of bug fixes and feature additions and interleave refactorings with extra semantic changes [25], [31]. Such extra edits are not always errors and could be intentionally made during refactoring. Nevertheless, according to a study at Microsoft, developers report that they would like to see semantic changes that deviate from pure refactoring separately, especially when checking refactoring correctness [25].

As shown in Figure 2, EEDETECTOR takes as input RefSet, which is generated by RefFinder. For each refactoring instance, it automatically applies an equivalent refactoring using a modified version of Eclipse's refactoring engine and creates a version with pure refactoring, P'. This version reflects what would be a correct and safe refactoring. Then, EEDETECTOR compares the generated pure refactoring version against the manual refactoring version, P_r. Differences between the two versions are highlighted.

Despite being a widely used refactoring tool, the Eclipse refactoring engine is not bug-free. Several studies found bugs in the Eclipse refactoring engine—the conditions under which refactoring generates compilation errors or behavioral changes [10], [23], [42]. Naively using the Eclipse refactoring engine to create a pure refactoring version can lead EEDETECTOR to false positives. To address this problem, EEDETECTOR first performs a checking step to avoid behavior-modifying refactorings. For each refactoring edit, if any of the bug conditions is matched, EEDETECTOR will not apply the automated refactoring and will instead report a warning message to the user. Table 2 summarizes the bug conditions checked by EEDETECTOR for a push down method refactoring; Eclipse generates unsafe refactoring for those scenarios. The first column indicates the bug number

from Eclipse's bug tracker,[4] and the second column shows a brief description of the bug condition where refactoring creates a compilation error or subtle behavioral change. Our technical report [7] describes all bug conditions for which EEDETECTOR checks, for the remaining five considered refactoring types.

It is important to highlight that EEDETECTOR might report refactoring problems that are not always real faults. However, we consider these to be still useful information as they are deviations from pure refactoring and deserve attention.

4 DETECTION CAPABILITY

For assessing REFDISTILLER's effectiveness in detecting anomalies, we perform two studies. We first use a data set with seeded refactoring anomalies to investigate its detection capability. In the second study, we assess REFDISTILLER's effectiveness and applicability in real scenarios using three open source projects by mining refactorings from their repositories. For both studies, we compare REFDISTILLER's results to regression testing as a baseline, as it is a de-facto method of validating refactorings in practice [25].

RQ1: *Is* REFDISTILLER *effective in detecting refactoring anomalies?*

In Sections 4.1 and 4.2, we discuss the details of each study. In Section 4.3, we discuss why it is difficult to directly compare REFDISTILLER to other related approaches in practice and include a theoretical comparison to GhostFactor [20], Ge et al.'s approach [21] and SafeRefactor [43].

4.1 Application to Seeded Anomalies

Data Set. To answer RQ1 we use a data set of one hundred refactoring transformations with seeded anomalies. Each transformation is a pair (p_1, p_2) of Java programs, where both p_1 and p_2 are free of compilation errors and p_2 contains at least one seeded refactoring anomaly that changes the behavior of the first version. The data set covers all six refactoring types and includes both missing edits (50) and extra edits (50).

The 50 seeded anomalies with missing edits are assembled from three sources. Most anomalies (38, 76% of missing edits) come from Soares et al.'s effort in detecting bugs in refactoring IDEs [42], [43]. In their work, they randomly generate small Java programs that are fed as input to Eclipse and other tools, making the applied refactoring yield programs with behavior distinct from the original —we assume these anomalies are as hard to detect manually. Therefore, Soares et al. are our study's main source of anomalies.

Missing edit anomalies are also established from the work by Cornélio et al. in defining formal, proved rules for refactoring [9]. Each rule specifies metaprograms which programs must match for the refactoring to be applied; also, rules can only be applied correctly when a number of preconditions are fulfilled by those programs. Although rules are based on a small theoretical language, it strongly resembles Java, and the preconditions in general are applicable, since they are conservative. For instance, the extract method rule (Rule 3.1 in Cornélio et al's work [9]) establishes several preconditions for a correct method extraction; one forbids the newly extracted method to be previously declared in any superclass or subclass of the current class. For our study, eight missing edit anomalies (16%) are built by intentionally skipping one of the preconditions—Anomaly EM4 [7], as an example, extracts the method as an overriding method from the superclass, violating the precondition.

Also, four missing edit anomalies (8%) are defined as variations from the guidelines of Fowler's refactoring book [17]. We apply mechanics (steps to be taken in manual refactoring) partially, by skipping one step that does not add compilation errors. For instance, Anomaly MM10 [7] is obtained by leaving out the call to the moved method—this call should be included as defined by one of the mechanics' steps.

Similarly, all extra edit anomalies are obtained by deliberately inserting random changes between steps from Fowler's mechanics [17]. Similar methodologies for inserting anomalies have been used in prior empirical studies on refactoring [8], [10], [20], [22], [43]. For example, Gligoric et al. systematically apply refactorings at a large number of places in well-known, open source projects and collected failures during refactoring or while trying to compile the refactored projects [22]. Two categories of changes are included: for the first, we add arbitrary statements to the body of methods affected by the refactoring—for instance, assigning a new value to an existing variable or updating an existing expression with a mathematical operation—`z=amount` replaced by `z=amount+10`; for the second, we change control flow by adding `if` statements or modifying their boolean condition.

Table 3 shows the distribution of seeded anomalies per refactoring type. Columns S, C, and F present the number of anomalies from Soares et al. [42], Cornélio et al. [9] and Fowler [17] respectively. Extra edit anomalies are split into A (changed or added statements) and B (updated boolean expressions). Our technical report includes a complete description of source code snippets and bug symptoms of each refactoring anomaly [7]. The dataset was assembled by the first author and later revised by two of the remaining authors.

TABLE 3
Distribution of seeded anomalies per refactoring type (**ME**: missing edit anomalies, **EE**: extra edit anomalies, **S**: the number of seeded anomalies which are collected from Soares et al.s' work, **C**: the number of seeded anomalies which are collected from Cornlio et al.'s work, **F**: the number of seeded anomalies which are adapted from Fowler's book, **A**: changed or added statements, **B**: changed boolean expressions, and **T**: the total number).

Refactoring Type	ME				EE		
	S	C	F	T	A	B	T
Extract Method	1	2	2	5	6	0	6
Inline Method	0	4	1	5	8	1	9
Move Method	8	1	1	10	6	3	9
Pull Up Method	10	0	0	10	9	0	9
Push Down Method	15	1	0	16	4	4	8
Rename Method	4	0	0	4	6	3	9
Total	38	8	4	50	38	12	50

Procedure. We compare REFDISTILLER against regression

4. The Eclipse bug tracker—https://bugs.eclipse.org/bugs/

testing, which is our baseline. We use an automated feedback-directed random test generation tool, Randoop[5] to create JUnit test suites [37]. Test suites created by Randoop have been successfully used to identify refactoring problems in prior work [28], [42], [43] and to find critical bugs in well-known refactoring engines (e.g., Eclipse and Netbeans). Although there are other test generation tools such as EvoSuite [18], a recent study [4] finds that, for the purpose of refactoring validation, EvoSuite's suites, in general, present similar or worse results when compared to Randoop's suites. To increase the variability of generated tests, a set of extra methods is added to the transformations independently. We run Randoop with a time limit of 100 seconds and the maximum test size of five statements. This configuration of 100 seconds was chosen by Soares et al. [41]. Soares et al. concluded that running Randoop for more than 90 seconds does not significantly increase test coverage rates for finding refactoring faults. Table 4 presents details regarding the generated test suites and corresponding test coverage at the level of statements and methods.

TABLE 4
Generated test suites and their coverages (**TS**: the number of test cases).

	TS	Coverage Rate	
		Statement	Method
Extract Method	11,228.7	63.6%	88.5%
Inline Method	8,568.4	59.1%	87.2%
Move Method	3,430.3	63.7%	89.0%
Pull Up Method	2,462.1	60.4%	86.2%
Push Down Method	3,807.0	62.7%	89.2%
Rename Method	3,268.4	64.5%	90.7%
Average	5,460.8	62.3%	88.5%

First, we run REFDISTILLER on each pair of the original and refactored program in our data set. We collect results for MEDETECTOR and EEDETECTOR and manually validate their outcomes to collect recall and precision. The validation process worked as follows. The first author analyzed results. Later, the results were validated in the meetings with two of the remaining authors. When there were disagreements, each case was put to a second analysis round and a joint decision was made.

Next, we run the regression test suite on both the original version and the refactored version. Any differences in the test outcomes are then considered as anomalies detected by regression testing.

We summarize the results using *Precision* and *Recall*, which are defined as follows: $TotF$ is the set of all injected refactoring anomalies; TP (true positives), the set of correctly identified anomalies; and FP (false positives), the set of incorrectly identified anomalies:

$$Precision = \frac{|TP|}{(|TP| + |FP|)}, Recall = \frac{|TP|}{|TotF|}$$

Results. Regarding recall, REFDISTILLER detects 96% of all seeded anomalies, outperforming regression testing by 21%. Table 5 summarizes the results. MEDETECTOR detects 47 anomalies out of 50, and EEDETECTOR detects 49 anomalies out of 50.

REFDISTILLER finds refactoring anomalies that are not easy to identify because they require understanding of complex code structures, e.g., overriding relationships in a deep class hierarchy. For instance, one of the callers of an inlined method is not updated after refactoring, referencing a different method with the same signature from its superclass at a higher level. This type of anomaly is seldom predicted when designing tests, generating subtle behavior changes that can easily pass unnoticed.

REFDISTILLER detects 24 anomalies that are not found by testing. Figure 3 shows a case in which tests fail to detect the anomaly. Lines 5-6 from `Calc.getVal` are extracted to `extrMeth`, and `if/else` statements are added to the new method (see Figure 3(b), underlined code). The extra edit changes the control flow depending on inputs. Although the generated test suite consists of more than 12,000 test cases with high test coverage (62.5% of statement coverage and 88.5% method coverage), this anomaly is not detected by test suites generated for the original version. The tests are inadequate to explore how extra edits can affect pre-existing methods or new ones and thus are incapable of exploring this new path. To identify the bug from Figure 3(b), the developer or the test generation tool must include a test case with a negative input to `getVal` in the old version.

It is important to highlight that REFDISTILLER complements testing. Although REFDISTILLER alone detects 96% of true seeded anomalies, 4 anomalies remain undetected. However, when both REFDISTILLER and testing are applied together, 3 additional anomalies are detected with 99% detection rate—100% for missing edits and 98% for extra edits. When applied alone, testing has a 75% detection rate only. Moreover, REFDISTILLER finds 24 anomalies not identified by testing, while testing alone identified only 3 anomalies not identified by REFDISTILLER.

False Negatives. 4 anomalies are not detected by REFDISTILLER (3 anomalies are not detected by MEDetector and 1 anomalies are not detected by EEDetector); they can only be detected by changing the state of an object/variable due to exception handling at runtime. Figure 4 exemplifies one of those false negatives. Lines 5-8 from `Element.m` are extracted into method `n`. In the original version, for `b` true, when the exception `e` is thrown at Line 7, method `m(boolean)` returns the state of `x` after the assignment `x = 23`. However, after extracting the method, when the same exception is thrown, the state of variable `x` is still `42` in the original method, which is then returned. This missing edit anomaly is hard to identify using MEDETECTOR since our current templates are not capable of capturing complex changes in the control flows due to incomplete refactoring. Heuristics to consider those scenarios are planned as future work.

Precision. We assess REFDISTILLER's precision by determining how many of the detected anomalies are indeed true anomalies. MEDETECTOR finds 63 missed edits, of which 47 are correct, resulting in 75% precision. EEDETECTOR finds 50 extra edits, of which 49 are correct. REFDISTILLER in total

5. http://randoop.github.io/randoop/

TABLE 5
Result of precision and recall (**SA**: the number of seeded anomalies, **M**: the number of anomalies detected by MEDETECTOR, **E**: the number of anomalies detected by EEDETECTOR, **T**: the number of anomalies detected by Testing, **TP**: the number of true positives, **P %**: precision, **R %**: recall, **M-T**: the number of anomalies detected only by MEDETECTOR, **T-M**: the number of anomalies detected only by testing, **M∩T**: the number of anomalies detected by both MEDETECTOR and testing, **M∪T**: a sum of anomalies detected by MEDETECTOR and testing, **E-T**: the number of anomalies detected only by EEDETECTOR, **T-E**: the number of anomalies detected only by testing, **E∩T**: the number of anomalies detected by both EEDETECTOR and testing, and **E∪T**: a sum of anomalies detected by EEDETECTOR and testing).

	MEDETECTOR					Testing		Comparison for Seeded Anomalies			
	SA	**M**	**TP**	**P %**	**R %**	**T**	**R %**	**M-T**	**T-M**	**M∩T**	**M∪T**
Extract Method	5	4	4	100%	80%	3	60%	2	1	3	5
Inline Method	5	5	5	100%	100%	3	60%	2	0	3	5
Move Method	10	10	9	90%	90%	10	100%	0	1	8	10
Pull Up Method	10	20	9	45%	90%	10	100%	0	1	9	10
Push Down Method	16	20	16	80%	100%	12	75%	4	0	12	16
Rename Method	4	4	4	100%	100%	4	100%	0	0	4	4
Sub Total	50	63	47	75%	94%	42	84%	8	3	39	50
	EEDETECTOR					**Testing**		**Comparison for Seeded Anomalies**			
	SA	**E**	**TP**	**P %**	**R %**	**T**	**R %**	**E-T**	**T-E**	**E∩T**	**E∪T**
Extract Method	6	6	6	100%	100%	5	83%	1	0	4	6
Inline Method	9	9	9	100%	100%	6	66%	3	0	6	9
Move Method	9	9	9	100%	100%	4	44%	5	0	4	9
Pull Up Method	9	9	9	100%	100%	8	88%	1	0	8	9
Push Down Method	8	8	7	87%	87%	4	50%	3	0	4	7
Rename Method	9	9	9	100%	100%	6	66%	3	0	6	9
Sub Total	50	50	49	98%	98%	33	66%	16	0	32	49
Total	100	113	96	85%	96%	75	75%	24	3	71	99

```
1  class Calc {
2   int getVal(int amnt) {
3    if (amnt > 10) {
4     int x = amnt + 10;
5     int z = x + 10;
6     return z;
7    } else {
8     int x = amnt * amnt;
9     return x; } }
10 }
```

(a) Original version.

```
1  class Calc {
2    int getVal(int amnt) {
3      if (amnt > 10){
4        int x = amnt + 10;
5  -     int z = x + 10;
6  -     return z;
7  +     return extrMeth(x);
8      } else {
9        int x = amnt * amnt;
10       return x; } }
11 +  int extrMeth(int x) {
12 +     if (x > 0)
13 +     return x + 10;
14 +     else return -1;
15 +  }
16 }
```

(b) Code after Extract Method refactoring with extra edits underlined.

Fig. 3. Anomaly detected by REFDISTILLER but not by testing.

finds 113 anomalies, of which 96 are correct, resulting in overall 85% precision.

Most false positives are due to incorrect identification of refactoring types by RefFinder. Although there are only one hundred refactorings seeded in the data set, RefFinder identified 32 additional refactoring instances. For instance, for a single pull up method refactoring, RefFinder reports an additional move method application. MEDETECTOR therefore runs two templates to check for potential refactoring missing edits, where it finds one false positive with respect to move method. For a push down refactoring with an extra edit, RefFinder reports three refactoring types: move method, extract method, and push down method. The change is moving method `calc` from class `A` to subclass `B`, turning the method call to delegation `B.calc`. When EEDETECTOR applies the push down method and move

```
1  class Element {
2    int m(boolean b) {
3      int x = 42;
4      try {
5       if (b) {
6         x = 23;
7         throw new Expt();
8       }
9      } catch (Expt e) {
10       return x; }
11     return x; }
12   int test() {
13     return m(true);}
14 }
```

(a) Original version.

```
1  class Element{
2    int m(boolean b) {
3      int x = 42;
4      try {
5  -     if (b) {
6  -       x = 23;
7  -       throw new Expt();
8  -     }
9  +     x = n(b, x);
10     } catch (Expt e) {
11       return x; }
12     return x; }
13 +  int n(boolean b, int x)
        throws Expt {
14 +     if (b) {
15 +       x = 23;
16 +       throw new Expt();
17 +     }
18 +     return x; }
19   int test() {
20     return m(true); }
21 }
```

(b) Code after a problematic Extract Method edit. Method `m` returns a different value when exception `e` is thrown due to a different of variable `x`'s state.

Fig. 4. Missing edit anomaly: Extract Method edits.

method refactorings, it correctly reports two types of extra edit errors. On the other hand, when it applies the false positive Extract Method refactoring with the Eclipse refactoring engine, it generates an incorrect edit. EEDETECTOR then attempts to compare the incorrect refactoring version with the manual version and subsequently reports a false positive extra edit.

Although 85% precision is high, it is lower than testing. Validation by testing does not generate any false positives (100% precision). However, it is important to emphasize that

although not generating false positives, testing misses several anomalies detected by REFDISTILLER. In a real scenario, undetected refactoring problems often result in software deterioration.

4.2 Evaluation using Open Source Projects

To further assess REFDISTILLER's effectiveness in real scenarios, we apply REFDISTILLER to three open source projects: XMLSecurity—a library for providing security functionality for XML such as authorization and encryption, JMeter—a performance analysis application for measuring performance under different load types, and JMock—a library for testing applications with mock objects.[6]

Data Set. For XMLSecurity and JMeter, we use data from prior work on refactoring change impact analysis [39], which identifies actual refactoring edits performed throughout several versions of these systems. The subject programs are drawn from the Software Infrastructure Repository (SIR) [12]. XMLSecurity contains 18 KLOC, while JMeter contains 32 KLOC. Since SIR provides release-level changes, we mine commit histories and map each refactoring edit to the earliest and closest commit revision. From this analysis we collect five pairs of versions. For JMock, we go through its repository history and search for commits with refactoring edits using RefFinder [38].[7] From this mining, we select three versions in which refactoring edits are found.

Study Results. Table 6 shows the number of missing and extra edits detected by REFDISTILLER: XMLSecurity (P1), JMeter (P2), and JMock (P3). Rn is each revision pair. We only found three available refactored versions of JMock, so R3, R4 and R5 do not apply.

Running REFDISTILLER takes on average 41.4, 205, and 26.5 seconds for the three projects respectively. We manually inspect individual anomalies to determine correcteness, and precision is 90%, 98%, and 100% for the three projects respectively. To calculate precision we use the same equation as described in Section 4.1.

We compare REFDISTILLER's anomaly detection capability with the error detection capability of existing JUnit regression tests included in SIR. The anomalies detected by REFDISTILLER and tests are later inspected by the authors. Table 8 presents the average number of existing test cases per system (considering versions) and their respective coverage rates. Though existing regression test suites are inadequate for covering refactoring edits, we use the existing tests as is to highlight the difficulty of validating refactoring edits without a static analysis technique in practice. These test suites detect only 2, 0, and 2 refactoring anomalies in the subject programs respectively. The results are aligned with the prior finding that only a small portion of refactoring edits are tested by existing regression tests and that refactoring mistakes might go unnoticed [5], [39].

Because REFDISTILLER uses a static analysis approach and focuses on the location of refactoring edits, it finds 184 additional anomalies not found by existing regression tests (Table 6). This number is found through manual inspection performed by the first author and later validated by the other authors independently. We cannot measure recall because we do not know the total number of all correct refactoring anomalies in these versions. Therefore, we measure the improvement by dividing the total number of REFDISTILLER's correct anomalies by the total number of anomalies found by testing. The overall improvement was 47 times, providing evidence that REFDISTILLER may effectively complement a testing-based approach for refactoring validation.

In a comparative analysis, Table 7 shows that, out of the 188 anomalies detected by REFDISTILLER, only 4 are identified by testing, and all four anomalies found by testing are also found by REFDISTILLER. Moreover, testing does not find any new anomaly. On the other hand, REFDISTILLER identifies 184 new anomalies neglected by testing.

Figure 5 exemplifies missing and extra edits detected by REFDISTILLER in XMLSecurity. In Figure 5(b), a developer extracted lines between 2 and 5 to the new method `canonicalizeTree` and made a mistake at line 13 by calling `cano.engineCanonicalizeTree(node)` instead of `cano.engineCanonicalize(node)`. MEDETECTOR detects this potential missing edit at line 13 due to a missed method reference binding to `engineCanonicalize(node)`. EEDETECTOR also finds potential extra edits at the same location at Line 13, because `cano.engineCanonicalizeTree(node)` is not part of the original method body extracted from Lines 3 to 5. REFDISTILLER shows the pure refactoring version in Figure 5(c) may help the developer when inspecting the refactoring.

False Positives. We manually inspect the results to investigate our false positive rate. As REFDISTILLER searches for refactoring deviations of a correct refactoring, we consider every code edition that is not related to a refactoring as a true anomaly. We did not contact the systems' developers to confirm whether those extra edits are in fact intended. Thus, our true positive rate might mask those scenarios. We discuss this threat to validity in Section 6.

After manual inspection, we observe a low false positive rate. In a few cases, false positives are caused by the combination of missing edits and extra edits at the same method location. For example, REFDISTILLER identified two extract method edits in the same method body. To obtain the pure refactoring version, EEDETECTOR subsequently applied two extract method refactorings to two different edit positions. However, the refactoring application to the first edit position affected the second refactoring edits, changing the second edit position. Applying multiple refactorings in the same method could prevent REFDISTILLER from locating extra edits correctly.

The results of both studies (with a seeded data set and with open source projects) help us to answer our first research question (RQ1). In both studies, REFDISTILLER detects several refactoring anomalies not detected by testing, the baseline error detection technique. Moreover, our study shows the analysis applied by REFDISTILLER does not take considerable time, and that it could be easily incorporated into a developer's daily routine without much overhead.

6. JMeter http://jmeter.apache.org/
XMLSecurity http://jmeter.apache.org/
JMock http://www.jmock.org/
7. https://github.com/jmock-developers/jmock-library/commits

TABLE 6
Results on open source projects: total correct anomalies found by REFDISTILLER is **188**, total correct anomalies found by existing regression tests is **4**, REDISTILLER precision is **94%** (188/200), and REFDISTILLER's improvement is **47** times (188/4) (**P1**: XMLSecurity, **P2**: JMeter, **P3**: JMock, **RE**: refactoring edits, **ME**: missing edits, **EE**: extra edits, **FP**: false positives, and **TM**: time (sec))

	REFDISTILLER															Testing		
	RE			ME			EE			FP			TM					
	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
R1	2	4	1	3	0	1	32	17	2	0	0	0	47	141	21	0	0	1
R2	2	2	4	9	3	2	5	12	0	10	2	0	43	127	32	0	0	1
R3	2	1	-	2	0	-	45	0	-	0	0	-	36	250	-	2	0	-
R4	1	1	-	0	0	-	1	8	-	0	0	-	39	235	-	0	0	-
R5	1	1	-	1	2	-	2	53	-	0	0	-	42	272	-	0	0	-
Total	22			200						12			Ave. 107			4		

TABLE 7
A comparative analysis: the number of anomalies detected by each or combined REFDISTILLER and testing techniques. **R** is the set of anomalies detected by REFDISTILLER and **T** the set of anomalies detected by Testing. **R - T** denotes detection results only by REFDISTILLER (**184**), **T - R** detection results only by Testing (**0**), **R ∩ T** detection results reported by both REFDISTILLER and Testing (4), and **R ∪ T** the number of anomalies reported by the combination of REFDISTILLER and Testing (**188**).

	R - T			T - R			R ∩ T			R ∪ T		
	P1	P2	P3	P1	P2	P3	P1	P2	P3	P1	P2	P3
R1	35	17	2	0	0	0	0	0	1	35	17	3
R2	4	13	1	0	0	0	0	0	1	4	13	2
R3	45	0	-	0	0	-	2	0	-	47	0	-
R4	1	8	-	0	0	-	0	0	-	1	8	-
R5	3	55	-	0	0	-	0	0	-	3	55	-
Total	184			0			4			188		

TABLE 8
Existing regression test suites and their coverage (**TS**: the average number of test cases).

	TS	Coverage Rate	
		Statement	Method
XML-Security	86.6	29.7%	26.6%
JMeter	14.6	3.0%	3.1%
JMock	186	89.2%	89.9%

4.3 Detailed Comparison of REFDISTILLER to Related Approaches

A few related approaches aim to detect refactoring anomalies by going beyond running existing regression tests. GhostFactor [20] checks correctness of manual refactoring, similar to MEDETECTOR, while Ge et al. [21] describes the concept of a refactoring-aware code inspection. SafeRefactor [42] validates automatic refactorings by generating random tests using Randoop and running the tests the versions before and after refactoring. However, we are unable to conduct direct comparison for practical reasons, such as language differences, or lack of tool availability.

Regarding Ghostfactor, its focus is on detecting missing edits only—extra edits are not addressed. The tool supports three refactoring types (extract method, inline method and change method signature). Two of them are dealt with by REFDISTILLER, while REFDISTILLER handles four refactoring types not supported by Ghostfactor (move method, rename method, push down method, push up method). We cannot directly compare REFDISTILLER with Ghostfactor, as it is a Visual Studio plug-in that works only for C#. As our data set consists of Java programs, to use GhostFactor, we would need to translate these programs from Java to C#. This task

```
1  byte[] canonicalize(Node node) throws
       CanonicalizeException {
2    if (node == null){
3      Log.error("Canonicalize a null node");
4    }
5    return cano.engineCanonicalize(node);
6  }
```

(a) The original code.

```
1  byte[] canonicalize(Node node) throws
       CanonicalizeException {
2  -   if (node == null){
3  -     Log.error("Canonicalize a null node");
4  -   }
5  -   return cano.engineCanonicalize(node);
6  +   return canonicalizeTree(node);
7  }
8
9  + byte[] canonicalizeTree(Node node) throws
       CanonicalizeException {
10 +   if (node == null){
11 +     Log.error("Canonicalize a null node");
12 +   }
13 +   return cano.engineCanonicalizeTree(node); // it
       should be 'return cano.engineCanonicalize(node);'
       instead.
14 + }
```

(b) The manual refactoring version.

```
1  byte[] canonicalize(Node node) throws
       CanonicalizeException {
2  -   if (node == null){
3  -     Log.error("Canonicalize a null node");
4  -   }
5  -   return cano.engineCanonicalize(node);
6  +   return canonicalizeTree(node);
7  }
8
9  + byte[] canonicalizeTree(Node node) throws
       CanonicalizeException {
10 +   if (node == null){
11 +         Log.error("Canonicalize a null node");
12 +   }
13 +   return cano.engineCanonicalize(node);
14 + }
```

(c) The pure refactoring version.

Fig. 5. The missing and extra edits detected by REFDISTILLER from versions of XMLSecurity. (a) The original code from XMLSecurity. (b) Code after an Extract Method edit in which REFDISTILLER detects a binding problem related to calling method `engineCanonicalize`. (c) Pure refactoring version generated by REFDISTILLER. It detects the underlined statement as an extra edit by comparing the pure refactoring version with the manual refactoring version.

is feasible yet problematic because our seeded anomalies are collected from prior work [9], [42] that are based on common errors found in Java refactoring tools.

Nevertheless, we perform a hypothetical study by manually applying the rules described in the Ghostfactor paper [20] to our data set of seeded missing edit anomalies (50). In our data set, there are ten missing edits related to either extract or inline method, refactoring types that can be handled by both GhostFactor and RefDistiller. GhostFactor would be able to detect five missing edits—three extract method and two inline method anomalies, while RefDistiller reveals nine missing edits—four extract method and five inline method anomalies. From the five anomalies detected by Ghostfactor, three are extract method, and two are inline method. For extract method, one is also missed by RefDistiller. However, RefDistiller detects an extra extract method anomaly missed by Ghostfactor. For inline method, RefDistiller detects all anomalies detected by Ghostfactor plus two extra cases that are missed by Ghostfactor. The anomalies missed by Ghostfactor are mainly related to statements not transferred, and/or references to methods/variables that are changed after refactoring (e.g., breaking or adding a overriding/overloading constraint). These properties are not checked by Ghostfactor.

RefDistiller goes further by pinpointing the location and cause of anomaly, such as *"Detected problematic binding in the reference* `getStorageData(value)`*, which may have affected the method* `Store.updateExpiration()` *- line 9"*. The value of this new information was tested during our user study to be detailed in Section 5. Professional developers stated that RefDistiller in fact improves and makes easier to detect and locate refactoring anomalies.

Regarding Ge et al.'s approach, a short paper describes the concept of a refactoring-aware code inspection [21]. However, to the best of our knowledge, no available implementation was found. It is important to note that Ge et al.'s approach would only detect extra edit anomalies, while RefDistiller is designed to detect both missing and extra edit anomalies. Like RefDistiller, Ge and Murphy-Hill leverage Eclipse refactoring APIs to separate pure refactorings. RefDistiller goes a step further by extending Eclipse refactoring APIs to prevent unsafe refactoring by checking bug conditions. This allows RefDistiller to apply automated refactoring in a safe manner, when isolating pure refactoring.

Finally, SafeRefactor validates automatic refactorings by generating random tests using Randoop and running the tests the versions before and after refactoring [42]. Therefore, it can be adapted for checking manual refactoring errors. Although we did not apply SafeRefactor as an alternative to RefDistiller, in our first study, we used randomly generated tests produced by Randoop [37], which essentially replicates the SafeRefactor implementation. We thus believe that our first study already compares against SafeRefactor.

5 User Studies

The previous two studies focus on evaluating RefDistiller's capacity of detecting refactoring anomalies. However, RefDistiller is designed not only for detecting refactoring anomalies but also for helping developers inspect refactorings. During maintenance tasks, developers must go beyond detection; they need to understand and locate problems in the codebase. In order to investigate both aspects and assess the practical benefits of using RefDistiller, we conduct a user study with professional developers. This study aims to evaluate RefDistiller's capacity of helping developers detect and locate refactoring problems. As a side goal, we evaluate RefDistiller's usefulness and its pros and cons compared to regression testing.

Participants Demographics. We recruit 15 active developers (14 male and 1 female) from different software companies (small and medium size), including roles as software developers, quality engineers, and project managers. Figure 6 summarizes the participants' background information. Most participants have experience with Java programming and the Eclipse IDE for at least three years. Most perform refactorings at least once a week or daily and, on average, have previous knowledge of 6.7 refactoring types from a given list of the 16 well-known refactoring types: add parameter, extract method, move method, push down method, decompose conditional, inline class, pull up field, rename method, encapsulate field, inline method, pull up method, replace parameter with a method, extract class, move field, push down field and substitute algorithm. 67% of the participants perform most of their refactorings manually.

Study Procedure. Prior to the study session, we asked the participants to answer a pre-study survey describing their demographic and background (Figure 6). After that, we held a seminar to go over the required background on the refactorings to be used during the study, and introduced RefDistiller's features and warning messages. This presentation included a fifteen-minute live demo on how to use RefDistiller built in the Eclipse IDE as a plug-in. Moreover, during the seminar, we clearly pointed out that RefDistiller could report possible false positives and/or false negatives. Therefore, we encouraged them to perform their own manual validation further.

Next, each participant received four pairs of versions of XMLSecurity's code. Each pair consists of a transformation with a refactoring-related anomaly, in which there was a single behavioral difference due to either a missing or extra edit. Then we asked the participants to inspect the refactorings by using two different approaches: regression testing (2 pairs), and RefDistiller (2 pairs). Both the task order and the tool assignment were randomly assigned for counterbalancing. It is important to highlight that all injected anomalies are detectable by both RefDistiller and testing, not particularly favoring any approach. The seeded anomalies are similar to the anomalies in our first study.

When inspecting each refactoring, the participants were asked whether the refactoring was problematic. If anomalies are detected, the participants should inspect the code and locate the anomaly by using the information given by one of the two methods: RefDistiller or testing. Each participant had 40 minutes to perform all four refactoring inspection tasks. During the study session, we encouraged them to verbalize their thoughts while performing the tasks. All study sessions and screen movements were recorded for further analysis. Finally, each participant completed a post-study survey to evaluate his or her experience. This survey

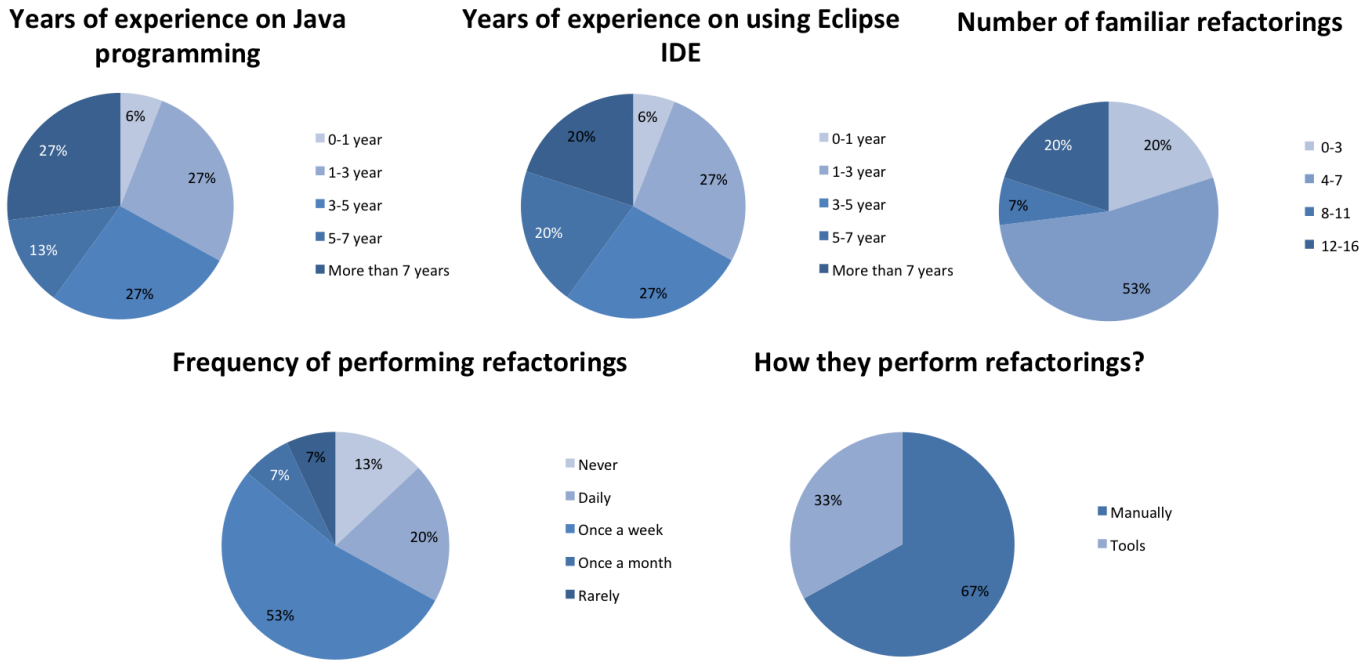

Fig. 6. Study participants' background information.

asked the participants to rate each refactoring inspection strategy on the *difficulty level* and *usefulness* for detecting and locating refactoring anomalies. Moreover, the survey included subjective questions in which the participants should list the pros and cons of each strategy. Pre- and post-study survey questionnaires and inspection task sheets are available on our website[8].

Inspection Tasks. We provided four inspection tasks. Each task covers a different refactoring edit: move method, extract method, rename method, and pull up method. An inspection task consisted of two versions of XMLSecurity's code, P and P', in which P' is P with an injected refactoring anomaly that changes P's behavior. They simulate anomalies that a developer might introduce when performing manual refactoring [8], [10], [20], [22], [43]. Figure 7 exemplifies one of the injected anomalies due to a binding problem. When moving method `engineCanonicalizeXPathNodeSet` from `Canonicalizer20010315Excl` to `Canonicalizer`, it breaks an existing overriding constraint (class `CanonicalizerBase` has a method `engineCanonicalizeXPathNodeSet` as well). As no compilation error is found, this anomaly might be hard to detect and locate. To inspect this refactoring, a developer needs to go through several classes, and carefully understand the system's class hierarchy and possible impact that this Move Method edit causes. To fix this anomaly, the developer would need to go through all callers of `engineCanonicalizeXPathNodeSet` and double-check whether the method reference is correct after the method is moved.

We asked the participants to inspect all four refactorings. No information was provided on whether the refactorings were safe or not. For each inspection task, we required the participant to use a designated inspection strategy (i.e., testing or RefDistiller) and provided an inspection sheet that included information regarding: (i) the two versions of XMLSecurity, before and after the refactoring (e.g., 7(a) and (b)); (ii) a description of the performed refactoring; and (iii) questions to be answered after the inspection was done. Table 9 shows the inspection sheet for the task described above. When using a testing method, developers were given XMLSecurity's existing test suite. This suite was collected from XMLSecurity's repository and comprises 103 JUnit test cases. Table 10 presents details about the test suite used in our study and the test coverage for the classes where anomalies appear.

Research Questions. To assess RefDistiller's capacity of helping developers to inspect refactorings, and to guide our investigation, we defined six research questions:

- RQ1: Does RefDistiller improve the detection of refactoring anomalies over regression testing?
- RQ2: Does RefDistiller improve the localization of refactoring anomalies over regression testing?
- RQ3: Is RefDistiller more useful for inspecting refactorings than regression testing?
- RQ4: Is RefDistiller more useful for locating refactoring anomalies than regression testing?
- RQ5: Does RefDistiller make refactoring inspection easier over regression testing?
- RQ6: Can developers save time in inspecting refactorings when using RefDistiller vs. regression testing?

Analysis on Improvement. From a total of 60 anomalies to be found (15 participants with 4 injected faults), half were inspected by using testing and the rest with RefDistiller. Only 70% of the faults were detected by the participants

8. https://sites.google.com/site/refdistiller/userstudy

TABLE 9
The description of a refactoring task and its corresponding questions for refactoring inspection.

Versions	Type	Description
350608.1 - 350608.2	Move Method	Method `engineCanonicalizeXPathNodeSet(Set)` was moved from `org.apache.security.c14.implementations.Canonicalizer20010315Excl` to `org.apache.security.c14.Canonicalizer`.
Q1. Was the refactoring edit semantics preserving, meaning the same inputs generate the same outputs before and after refactoring? (Yes/No) Q2. If you have answered NO in Q1, try to locate the bug and describe it.		

```
1  public class Canonicalizer20010315Excl extends
       CanonicalizerBase {
2    public byte[] engineCanonicalizeXPathNodeSet(Set
       xpathNodeSet)
3        throws CanonicalizationException {
4      return this.engineCanonicalizeXPathNodeSet(
       xpathNodeSet, "");
5    }
6  }
7  public class CanonicalizerBase ...{
8    public byte[] engineCanonicalizeXPathNodeSet(Set
       xpathNodeSet)
9      throws CanonicalizationException {...}
10 }
11 public class Canonicalizer {...}
```

(a) The original code.

```
1  public class Canonicalizer20010315Excl extends
       CanonicalizerBase {
2  -   public byte[] engineCanonicalizeXPathNodeSet(Set
       xpathNodeSet)
3  -       throws CanonicalizationException {
4  -     return this.engineCanonicalizeXPathNodeSet(
       xpathNodeSet, "");
5  -   }
6  }
7  public class CanonicalizerBase ...{
8    public byte[] engineCanonicalizeXPathNodeSet(Set
       xpathNodeSet)
9      throws CanonicalizationException {...}
10 }
11 public class Canonicalizer {
12 +   public byte[] engineCanonicalizeXPathNodeSet(Set
       xpathNodeSet)
13 +       throws CanonicalizationException {
14 +     return this.engineCanonicalizeXPathNodeSet(
       xpathNodeSet, "");
15 +   }
16 }
```

(b) Code with an injected refactoring anomaly.

Fig. 7. Example of injected missed edit anomaly used in our user study. When moving `engineCanonicalizeXPathNodeSet` from `Canonicalizer20010315Excl` to `Canonicalizer`, we break an overriding constraint, since `CanonicalizerBase` has a method `engineCanonicalizeXPathNodeSet` as well.

using regression testing. On the other hand, all 30 were detected by REFDISTILLER. Participants were asked to localize the anomalies. They correctly pinpointed the location of the anomaly by using REFDISTILLER in 90% of the cases, while by using testing this rate dropped to only 13%. Figure 8 presents a comparative view of these data. We performed a Chi-Squared proportion test to verify whether the found differences between the rates of fault detection and localization are statistically significant. Table 11 shows the proposed null hypothesis and the alternative hypothesis in both tests, and the corresponding X-squared and p-values. With 95% confidence, the tests found significant differences in both analyses, i.e., we rejected both null hypotheses. Thus, we can answer RQ1 and RQ2 as *there is evidence that using* REFDISTILLER*, developers detected and located more refactoring anomalies than by using tests*.

TABLE 10
Information about the test suite provided to the participants and the coverage levels of the classes where the faults were injected per inspection task.

XML-Security

Number of test cases	Statement coverage	Method coverage	Inspection tasks	Coverage of edited class (statement coverage)
103	31.6%	30%	**task 1**	79.5%
			task 2	44.6%
			task 3	48.8%
			task 4	32.1%

To investigate whether our results were biased by the participants' experience with programming or refactoring, we related the cases where the anomalies were detected/located to participants' level of experience and how often they perform refactorings. Figure 9 shows a few histograms. As can be seen, when using testing, the rates of both anomaly detection and localization seem to increase as the participants' experience level is higher. Inspecting a refactoring with tests only seems to be hard. Inexperienced developers tend not to have a deep understanding of what the pitfalls are and the possible impact of a specific refactoring. More experienced developers might have used their previous knowledge to better understand the test suite's execution messages and to search the right code elements when locating the problems. On the other hand, REFDISTILLER's results seem to be consistent, regardless of developer's experience level. For instance, less experienced developers in our study (participants who stated that they have never performed refactorings) were able to detect and locate both anomalies with REFDISTILLER but none using testing. The following quote was collected from a participant while using REFDISTILLER:

"RefDistiller enabled me to find the problems that would not be easily found by naked eye or by testing, even for an expert developer."

Sometimes, although there were failing tests, some participants reported no anomalies. This fact shows that developers often do not trust test results, since they cannot assess with confidence the quality of a test suite. This fact was also raised by prior research [39].

Analysis on Usefulness. To assess developers' perspective regarding the usefulness of REFDISTILLER versus testing for inspecting refactorings, we asked them to rate both strategies among one of the five options: *not useful*,*little useful*, *neutral*, *useful* or *very useful*. Figure 10(a) presents

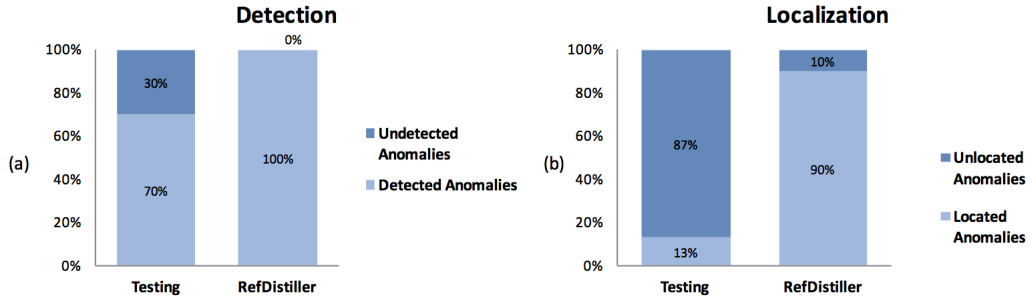

Fig. 8. Rates of anomaly detection and localization.

TABLE 11
Null and alternative hypothesis regard the proportion tests. D is the rate of detected anomalies and L is the rate of located anomalies.

H0	H1	X-Squared	p-value
$D_{testing} = D_{RefDistiller}$	$D_{testing} \neq D_{RefDistiller}$	8.366	0.003823
$L_{testing} = L_{RefDistiller}$	$L_{testing} \neq L_{RefDistiller}$	32.3026	1.319e-08

the histograms corresponding to this analysis. While most participants found testing to be *little useful* for inspecting refactorings, most found REFDISTILLER *useful* or *very useful*. In a similar analysis, we asked them to rate both strategies on their usefulness regarding providing helpful information to locate the anomalies (Figure 10(b)). Most participants found testing information not useful for locating the refactoring problems, while most participants found REFDISTILLER's messages *useful* or *very useful* for locating the problems. The following quotes were collected from participants of our study:

"Junit tests help finding bugs but do not give much help on locating them. REFDISTILLER*'s way of displaying bugs is more informative and very intuitive."*
"Failing test cases and their messages do not give me the needed background for understanding the reason of the bug nor to locate it."
"The combination of missing and extra edits seems to be a powerful way of expressing refactoring bugs."
"I see the extra edits view as a useful tool because it can show a possible way for fixing the found refactoring bug."

Thus, we can answer RQ3 and RQ4 by saying that, for the context of our study, *developers find* REFDISTILLER *useful for inspecting refactorings and that it provides more help in locating refactoring problems than regression testing*.

Analysis on Difficulty. We asked participants to rate each strategy regarding the difficulty level of inspecting refactorings by using testing or REFDISTILLER. Figure 11 presents the histogram of this analysis. While most participants rate the difficulty level of inspecting refactorings with REFDISTILLER as *easy* or *very easy*, they rate the difficulty level of using testing for the same task as hard or medium. A developer stated:
"Inspect a refactoring is really hard, unless there is a very effective test suite to help. However, the help that REFDISTILLER *brings to locate the bugs makes this task way easier."*
Thus, we can answer R5 and state that *it is easier to inspect refactorings using* REFDISTILLER *than with testing*.

Analysis of time improvement. From the recordings of participants' inspection sessions, we measured the time participants spent when performing the tasks. On average, developers spent 7.16 minutes for inspecting a refactoring with testing, while they spent 5.9 minutes with REFDISTILLER. Thus, we can answer RQ6 and say that REFDISTILLER *helped developers to save, on average, 17% of time over testing*. Although this rate might not be very impressive, when we relate the time spent when using each strategy with the number of detected and located anomalies, we can see that the time spent when using REFDISTILLER was more productive (100% anomalies detected and 90% anomalies located). It is important to highlight that the time computed for each task includes REFDISTILLER's running time. As REFDISTILLER applies a static analysis, its running time is often higher than running XMLSecurity's test suite. To emphasize our conclusions, here are some quotes collected regarding time:
*"*REFDISTILLER *would definitely save me time when inspecting my refactored code. Sometimes we spend a long time trying to find small mistakes that can easily pass unnoticed."*
"There is a practical need for tools like REFDISTILLER *(that accelerate the process of detecting and finding bugs during software development) because nowadays there is an urge for faster software releases, which often lead to buggy software."*

Subjective Analysis and Discussion. Finally, through post survey questions, we asked developers to evaluate both testing and REFDISTILLER. They listed the pros and cons of each inspection strategy and described their general perception of using REFDISTILLER in comparison with testing. In general, the participants found testing a valid strategy for detecting refactoring problems but not useful when the goal is to locate the cause of the problem. Testing messages are reported as not informative and not useful for neither bug understanding or localization. However, one participant stated:
"A single problem might lead to several failing test cases. This redundant data might be an interesting source of information when tracking a bug."

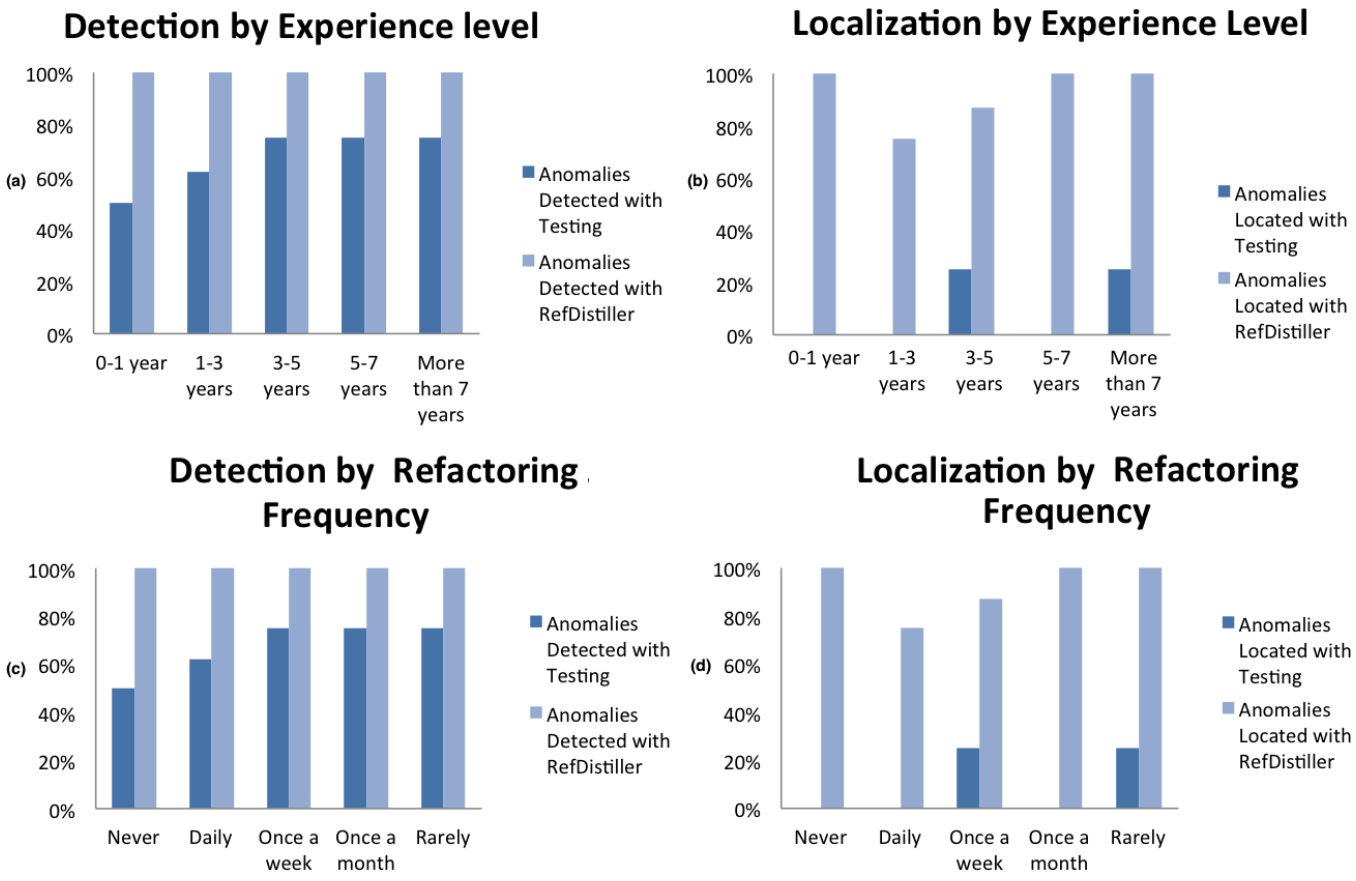

Fig. 9. Detection and localization rates by participants' experience with programming and frequency of refactoring levels.

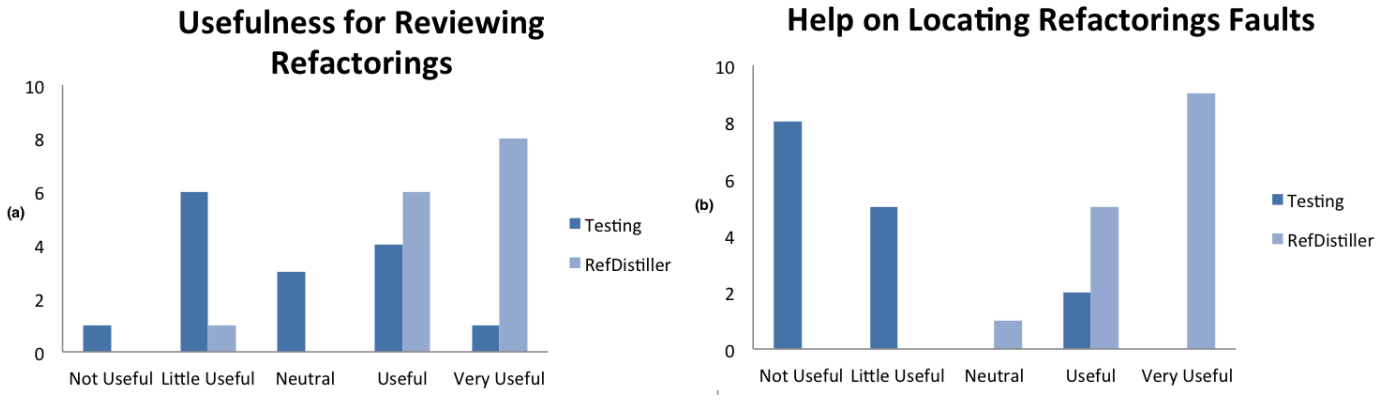

Fig. 10. Analysis regarding participants' perception on (a) usefulness of testing and REFDISTILLER for inspecting refactorings and (b) its help on locating refactoring anomalies.

Although that is an interesting point of view, other participants complained about the number of failing test cases. For instance:
"Most of the time I got confused when locating the bugs with testing. Several test cases failed and pointed to different parts of the code, which made debugging harder."

Participants listed several practical benefits of using REFDISTILLER, such as productivity improvements and cut-back of inspection time. They found REFDISTILLER more informative and effective for inspecting refactorings. Most of them found REFDISTILLER's output messages clear and helpful. The way REFDISTILLER helps fault localization was a widely positive point. They emphasize that, by pinpointing the location where the refactoring problem might be, and a possible reason, REFDISTILLER made the inspection process easier to perform. Moreover, participants highly appreciate the extra edits view. By showing a pure-refactoring version, developers can better visualize their mistakes and come up with possible ways of fixing the mistakes. The following are

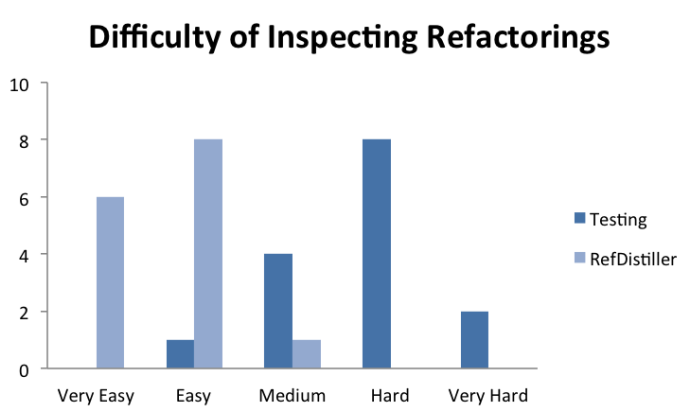

Fig. 11. Analysis regarding rate on difficulty level of inspecting refactorings by using testing or REFDISTILLER.

quotes about the extra edits view:
"The extra edits view seems to be very helpful for creating patches."
"The extra edits found are a great source of refactoring bad smells."

Participants also listed some usability issues and provided suggestions for improving REFDISTILLER. For instance, a participant mentioned that some warning messages in the missing edits view could be improved.
"The missing edits warning messages were too vague, which took me a long time to relate the reported problem with my code."
He suggested more specific messages and a more fine-grained class of anomalies. Another participant suggested the introduction of a merge option in the extra edits view. This option would take the manual and the pure-refactoring versions, merge them, and suggest a fixing patch. One developer found the time waiting for REFDISTILLER to run too long. However, several other participants emphasized REFDISTILLER's benefits in time saving. All these suggestions will be considered in the future evolution of REFDISTILLER.

One participant rated REFDISTILLER as *not very useful*. This participant is used to performing refactoring inspection with a different code *diff* tool. As we asked him to inspect using only the designed strategy (REFDISTILLER or testing) he ended up spending more time in the beginning, since he was prevented from using the diff tool he is familiar with. However, the same participant pointed out the benefits of REFDISTILLER's new features, especially providing the location of the detected refactoring problem. He also mentioned that he would easily incorporate REFDISTILLER into his current tool set.

Some participants provided an interesting practical scenario for using REFDISTILLER. They indicated that testing lacks useful information for locating refactoring bugs; however, they still trust tests as not presenting false positives. Thus, these participants mentioned that, in practice, they would combine testing and REFDISTILLER. They would first run their test suite and collect problems, then run REFDISTILLER to detect problems that might have passed unnoticed by the testing validation. Finally, they would use REFDISTILLER to locate all bugs found by both strategies and fix them. We believe this is a good usage scenario of our tool and reinforces our idea that REFDISTILLER complements testing.

6 THREATS TO VALIDITY

Regarding studies on detection, in terms of **construct validity**, the accuracy of RefFinder's refactoring reconstruction directly affects REFDISTILLER's capability in detecting refactoring faults. We refer to Prete et al. for further evaluation data on how varying its similarity threshold affects RefFinder's accuracy [38]. When multiple interfering refactorings are seeded in the same version together, REFDISTILLER may find false positives or negatives. For example, when we seed the Extract Method and Inline Method refactorings in the same location, which partially cancels each other, REFDISTILLER finds two separate faults that should not be found—one related to the Extract Method and the other related to the Inline Method. Moreover, as discussed in the *False Positives* subsection, EEDETECTOR's false positive rate tends to be higher. However, we believe that any extra edits in the refactoring inspection should be confirmed, since those extra edits might interfere with software's original behavior. Professional developers stated the same concern [25], [48].

Since REFDISTILLER applies a static analysis, it is currently unable to detect runtime object types precisely and cannot capture control flow changes in the exception handling logic precisely.

REFDISTILLER takes the old and new version snapshots as input and does not leverage edit history during programming sessions (e.g., a sequence of edit operations recorded in Eclipse). Extending and applying REFDISTILLER to edit histories or other richer refactoring data [33], [45] remains as future work.

In terms of **internal validity**, we detect refactoring anomalies in which behavioral changes occur. Not all identified issues are indeed anomalies and could be intentional. Nevertheless, we believe that paying attention to behavior changes occurred during refactoring is worthwhile.

In terms of **external validity**, our results do not generalize beyond our data set and the three subject programs. In our evaluation, REFDISTILLER was able to detect anomalies not found by either existing test suites or an automated test generation tool, Randoop. However, a test suite's refactoring validation power is relative to several factors that might influence its effectiveness, such as coverage level, or variability of data. In the empirical study, we evaluated REFDISTILLER with real regression test suites, but some of the test suites may be inadequate for evaluating REFDISTILLER. For example, JMeter includes only 14.6 test cases with 3% code coverage on average. Despite its low coverage rate, our results empirically showed the benefits of static analysis and the challenges of testing for validating refactoring edits in practice. Therefore, the potential gain of using REFDISTILLER, instead of testing, might be relative; however, we believe that the combination of both strategies—testing and REFDISTILLER—is likely to allow developers to better inspect refactorings.

Finally, the injected anomalies do not cover all possible refactoring-related bugs. However, these faults were collected from previous studies that identified problems in

automated refactoring tools and/or inspired by real scenarios the authors experienced when performing/inspecting refactorings. Regarding possible subjectivity bias, the results of both studies were collected by the first author and later inspected by two other authors in an independent manner.

Regarding the user study, we discuss the following threats to validity.

In terms of **construct validity**, we used multiple-choice questions aspects such as usefulness and help for detecting/locating problems. Objective measures, such as time spent browsing through classes irrelevant to refactoring problems, could be used to measure the same aspects. However, our goal was to see the practical impact that each inspection strategy has on developers' opinions. In addition, our study comprises different refactoring types and anomalies, and we counterbalanced the order and task assignment to mitigate the learning effects.

In terms of **external validity**, we selected participants from different projects, with different roles and levels of experience. However, besides the time spent in our study, they did not have any familiarity with XMLSecurity's code. Thus, one may argue that our study's conclusions may not generalize to professional developers who have familiarity with their codebase. However, we do not believe this is a significant limitation. XMLSecurity is a medium size Java project that can be a representative subject for a Java application. Thus, although they may not have been familiar with XMLSecurity's code specifics, we believe developers were familiar with the the general aspects of a Java application. Moreover, the refactorings applied, the injected anomalies and their impacts were similar to common scenarios when refactoring/inspecting any Java program.

Another threat is that the tasks we selected included anomalies that RefDistiller can detect. Other refactoring anomalies might lead to different conclusions. However, we believe that our tool is still useful as the refactoring problems detected by RefDistiller are shown to be frequent [19], [42].

Finally, we used only XMLSecurity's test suite, and thus we cannot generalize the results to all regression test suites. However, XMLSecurity's suite was extracted from an open source project. Moreover, versions of the XMLSecurity's suite have been used in other testing-related empirical studies (e.g. [6], [13], [49]). Therefore, by using real systems, by encouraging the need for inspection, by using the actual test suites available, and by considering only kinds of anomalies reported by other authors, we believe the study resembles a realistic setting.

To mitigate **internal validity**, before the participants started the inspection tasks, we introduced them to RefDistiller and its features through a presentation and tool demo. Moreover, the first author was available to help participants with any usability issue during all study sessions.

Another threat is a possible social desirability bias. Participants' perceptions could have been more positive towards RefDistiller because they knew the authors created it. Although this threat is hard to mitigate, we tried to avoid positive bias by mentioning during the pre-session seminar that RefDistiller could report false positives or negatives. In fact, we noticed that some participants spent a considerable amount of time deciding whether to trust RefDistiller and double-checking its outputs.

7 Related Work

Opdyke and Johnson [34] create the term refactoring and formally define refactorings as (1) generalizing an inheritance hierarchy, (2) specializing an inheritance hierarchy, and (3) using aggregations to model the relationships among classes. Fowler creates a catalog of different refactoring types and defines steps to proceed with the required edits in terms of mechanical descriptions [17].

Popular IDEs such as Eclipse, IntelliJ, NetBeans, and Visual Studio include support for automated refactoring. Those tools check pre- and post-conditions to prevent refactoring anomalies. However, recent studies show that automated refactoring is underused in practice. Murphy et al. find that 90% of the refactoring edits are done manually [31]. Negara et al. find that developers perform manual refactoring more often than automated refactoring [32]. Kim et al. find that 51% of developers perform 100% of their refactoring manually [25]. Other studies reflect on reasons for this underuse, such as usability issues [30], unawareness [46], and a lack of trust [10], [28], [42], [46].

Refactoring is error prone. Dig et al. state that over 80% of the API changes that break existing applications are refactorings [11]. Weißgerber and Diehl find a correlation between refactoring edits and an increasing number of bug reports [47]. Kim et al. report that there is an increase in the number of bug fixes after API-level refactorings [24].

Formal verification is an alternative for avoiding refactoring anomalies [26]. Cornélio et al. propose rules for guaranteeing semantic preservation [9]. Similarly, Mens et al. use graph rewriting for specifying refactorings [27]. Overbey et al. present a collection of refactoring specifications for Fortran 95 [35]. However, these approaches focus on improving the correctness of automated refactoring through formal specifications, as opposed to finding anomalies during manual refactoring.

Daniel et al. propose a technique for testing refactoring engines by automatically generating input programs [10]. Gligoric et al. apply refactoring edits systematically to find bugs in Eclipse refactoring [22]. Overbey and Johnson propose a differential conditional checker to be used by refactoring engines for checking behavior preservation [36]. Schaeffer and Moor verify whether there is behavior preservation by comparing data and control dependencies in the original and refactored code [40]. These approaches differ from RefDistiller by improving the correctness of automated refactoring as opposed to finding errors during manual refactoring.

Regression testing is the most used strategy for checking refactoring correctness. However, Rachatasumrit and Kim find that test suites are often inadequate and developers may hesitate to initiate or perform refactoring tasks due to inadequate test coverage [25], [39]. Soares et al. design and implement SafeRefactor that uses randomly generated test suites for detecting refactoring anomalies [43]. Mongiovi et al. introduces SafeRefactorImpact [28]. SafeRefactorImpact extends SafeRefactor by adding an impact analysis

step. SafeRefactorImpact decomposes an edit into small-grained transformations and analyzes the impact of each one. Then, it uses Randoop to generate test cases for the impacted methods. In our studies, we show that even tool-generated tests can be inadequate. Using a SafeRefactor like testing validation, we find that about 25% of the refactoring anomalies are not identified by using generated test suites, even with a long generation time limit (100 seconds). Moreover, SafeRefactorImpact impact rules and REFDISTILLER's refactoring templates have different goals and applications. While SafeRefactorImpact identifies impacted elements that are used for guiding a testing generation process, REFDISTILLER aims to verify whether steps for a successful refactoring are indeed performed.

GhostFactor checks the correctness of manual refactoring [20], similar to MEDETECTOR. However, unlike EEDETECTOR, GhostFactor does not have any capability to isolate potential behavior changes from pure refactoring by running an equivalent automated refactoring. GhostFactor detects missing edits only, while REFDISTILLER detects both missing and extra edits. Although we did not run GhostFactor for comparison, according to GhostFactor's algorithm, it supports only three refactoring types and detects only potential missing edits. Our analysis showed that GhostFactor was able to detect only three missing edits from the transformations collected from Soares work [42], as opposed to 34 correct missing edits found by REFDISTILLER. Moreover, when applied to the open-source dataset, it found five missing edits, while REFDISTILLERfound 24. It is important to highlight GhostFactor finds only missing edits anomalies and only for three refactoring types.

REFDISTILLER goes further than all tools listed above and traditional testing validation by pinpointing the location and cause of an anomaly, such as *"Detected problematic binding in the reference* `getStorageData(value)`*, which may have affected the method* `Store.updateExpiration()` *- line 9"*. The value of this new information was tested during our user study. Professional developers stated that REFDISTILLER in fact improves and makes it easier to detect and locate refactoring anomalies.

Ge and Murphy-Hill propose a refactoring-aware code review tool (a four-page workshop paper published in Spring 2014 [21], with goals similar to REFDISTILLER). This tool helps reviewers by identifying applied refactorings and letting developers examine them in isolation. Like REFDISTILLER, Ge and Murphy-Hill leverage Eclipse refactoring APIs to separate pure refactorings. REFDISTILLER goes a step further by extending Eclipse refactoring APIs to prevent unsafe refactoring by checking bug conditions. This allows REFDISTILLER to apply automated refactoring in a safe manner when isolating pure refactoring. In addition, REFDISTILLER also detects missing edits (incomplete refactorings) with concrete warning messages about how to fix the anomalies. Some refactoring anomalies are hard to detect, and an analysis that searches for both missing and extra edits might provide a better understanding of the problem.

Ge at al. perform a survey with professional developers to investigate the role of refactoring-aware code review in practice [48]. This survey highly motivates our work. It shows that reviewing a refactoring is a real concern for developers and that it requires much effort and dedication, which are issues that we address in REFDISTILLER.

Tsantalis and Chatzigeorgiou propose an approach and tool (JDeodorant) for identifying refactoring opportunities [15], [44]. They also introduce a set of rules regarding the preservation of existing dependences. Different from MEDETECTOR's refactoring templates, JDeodorant's rules are designed to predict whether a possible future refactoring may change the behavior. On the other hand, MEDETECTOR's templates check whether edits are performed according to expected refactoring edits.

Ge and Murphy-Hill [19] and Foster et al. [16] detect manual refactoring, remind a developer that automated refactoring is available, and complete it automatically. While these refactoring completions tools leverage Eclipse refactoring APIs, REFDISTILLER differs from these by finding anomalies as opposed to auto-completing refactorings.

8 CONCLUSION

Developers often mix refactoring with other semantic changes, and they do most refactoring manually. This manual execution often leads to defect inclusion. During maintenance tasks, developers need to validate their refactorings and, locate and fix it when a problem is found. We present REFDISTILLER, a tool for detecting and locating anomalies in manual refactoring. To detect missing edits, we use predefined change rules for each refactoring type and match expected constituent edits with actual syntactic edits. To detect extra edits that may change a program's behavior, we extend and leverage existing Eclipse refactoring APIs to separate pure refactoring from behavior-modifying edits. While such extra edits are not always errors and could be intentional, we believe that isolating them can help developers focus on their attention to semantic changes.

The evaluation shows that our static analysis approach can effectively complement testing-based refactoring validation. REFDISTILLER finds anomalies that the existing test suites falls short of finding. Our study with professional developers shows that REFDISTILLER is effective and helps developers to better inspect refactorings. To the best of our knowledge, our work is the first to detect both missing edits and extra edits that deviate from pure refactoring and to provide useful information for fixing refactoring problems. Its capability can help developers focus on their attention to subtle behavior changes with no compilation errors.

As future work for improving REFDISTILLER's approach and tool, we intend to: (i) create refactoring templates for more refactoring types; (ii) make warning messages even more specific and fine-grained; and (iii) implement a merge operation in the extra edits view to propose a way of fixing found problems based on the manual and pure-refactoring versions.

9 ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under grants CCF-1527923, CCF-1460325, CNS-1239498, a Google Faculty Award, and by the National Institute of Science and Technology for Software Engineering, funded by CNPq/Brasil, grant 573964 /2008-4.

REFERENCES

[1] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software inspections: an effective verification process. *IEEE software*, (3):31–36, 1989.

[2] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau. Software inspections and the industrial production of software. In *Proc. of a symposium on Software validation: Inspection-testing-verification-alternatives*, pages 13–40. Elsevier North-Holland, Inc., 1984.

[3] E. Alves, M. Song, and M. Kim. Refdistiller: A refactoring aware code review tool for inspecting manual refactoring edits. In *International Symposium on Foundations of Software Engineering, Research Demonstration Track*, page 4, 2014.

[4] E. L. Alves, P. D. Machado, T. Massoni, and M. Kim. Prioritizing test cases for early detection of refactoring faults. *Software Testing, Verification and Reliability*, 2016.

[5] E. L. Alves, T. Massoni, and P. D. de Lima Machado. Test coverage of impacted code elements for detecting refactoring faults: An exploratory study. *Journal of Systems and Software*, 2016.

[6] E. L. Alves, T. Massoni, and P. D. Machado. Test coverage and impact analysis for detecting refactoring faults: a study on the extract method refactoring. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1534–1540. ACM, 2015.

[7] E. L. Alves, M. Song, M. Kim, P. D. Machado, and T. Massoni. Additional artifacts for "refactoring inspection support for manual refactoring edits". Technical report, The University of Texas at Austin, 2016. https://sites.google.com/site/refdistiller/TechnicalReport.pdf.

[8] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 104–113. IEEE, 2012.

[9] M. Cornélio, A. Cavalcanti, and A. Sampaio. Sound refactorings. *Science of Computer Programming*, 75(3):106–133, 2010.

[10] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 185–194. ACM, 2007.

[11] D. Dig and R. Johnson. The role of refactorings in api evolution. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 389–398. IEEE, 2005.

[12] H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 60–70. IEEE, 2004.

[13] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *15th International Symposium on Software Reliability Engineering, (ISSRE)*, pages 113–124. IEEE, 2004.

[14] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

[15] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, 2012.

[16] S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 222–232. IEEE Press, 2012.

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

[18] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[19] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *34th International Conference on Software Engineering (ICSE)*, pages 211–221. IEEE, 2012.

[20] X. Ge and E. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *Software Engineering (ICSE 2014) 36th International Conference on*. IEEE, 2014.

[21] X. Ge, S. Sarkar, and E. Murphy-Hill. Towards refactoring-aware code review. In *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 99–102. ACM, 2014.

[22] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov. Systematic testing of refactoring engines on real software projects. In *ECOOP 2013 - Object-Oriented Programming*, pages 629–653. Springer, 2013.

[23] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in udita. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 225–234. ACM, 2010.

[24] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160. ACM, 2011.

[25] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 50. ACM, 2012.

[26] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[27] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.

[28] M. Mongiovi, R. Gheyi, G. Soares, L. Teixeira, and P. Borba. Making refactoring safer through impact analysis. *Science of Computer Programming*, 2013.

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

[30] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring. In *ICSE'08. ACM/IEEE 30th International Conference on Software Engineering*, pages 421–430. IEEE, 2008.

[31] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[32] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *ECOOP 2013–Object-Oriented Programming*, pages 552–576. Springer, 2013.

[33] S. Negara, M. Vakilian, N. Chen, R. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In J. Noble, editor, *ECOOP 2012 Object-Oriented Programming*, volume 7313 of *Lecture Notes in Computer Science*, pages 79–103. Springer, Berlin, Heidelberg, 2012.

[34] W. F. Opdyke. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proc. 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, 1990.

[35] J. L. Overbey, M. J. Fotzler, A. J. Kasza, and R. E. Johnson. A collection of refactoring specifications for fortran 95. In *ACM SIGPLAN Fortran Forum*, volume 29, pages 11–25. ACM, 2010.

[36] J. L. Overbey and R. E. Johnson. Differential precondition checking: A lightweight, reusable analysis for refactoring tools. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 303–312. IEEE, 2011.

[37] C. Pacheco and M. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications*, pages 815–816. ACM, 2007.

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

[39] N. Rachatasumrit and M. Kim. An empirical investigation into the impact of refactoring on regression testing. In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 357–366. IEEE, 2012.

[40] M. Schaefer and O. De Moor. Specifying and implementing refactorings. In *ACM Sigplan Notices*, volume 45, pages 286–301. ACM, 2010.

[41] G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, and T. Massoni. Analyzing refactorings on software repositories. In *Software Engineering (SBES), 2011 25th Brazilian Symposium on*, pages 164–173. IEEE, 2011.

[42] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[43] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *Software, IEEE*, 27(4):52–57, July 2010.

[44] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *Software Engineering, IEEE Transactions on*, 35(3):347–367, 2009.

[45] M. Vakilian, N. Chen, S. Negara, B. Rajkumar, R. Zilouchian Moghaddam, and R. Johnson. The need for richer refactoring usage data. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, pages 31–38. ACM, 2011.

[46] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 233–243, 2012.

[47] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *Proceedings of the 2006 international Workshop on Mining Software Repositories*, pages 112–118. ACM, 2006.

[48] J. W. Xi Ge, Saurabh Sarkar and E. Murphy-Hill. Refactoring-aware code review. Technical report, North Carolina State University, 2014. Paper under revision. Please contact the authors for a copy.

[49] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. An empirical study of junit test-suite reduction. In *IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 170–179. IEEE, 2011.

Everton L. G. Alves is a doctoral candidate in the Computing and Systems Department at Federal University of Campina Grande (UFCG), Brazil. He received a Master degree in Computer Science from the Federal University of Campina Grande, Brazil, in 2011 and a Bachelor degree in Computer Science in 2009, from the same university. Between 2013/2014 he was a visiting student at the Department of Electrical and Computer Engineering at the University of Texas at Austin under the supervision of PdD Miryung Kim. His main interests include software maintenance, regression testing, model-driven development and testing, real-time systems and integration.

Myoungkyu Song is an assistant professor at the Computer Science Department at the University of Nebraska at Omaha since 2015. Prior to coming to UNO, he was a postdoc in the Center for Advanced Research in Software Engineering (ARiSE) at the Department of Electrical and Computer Engineering at the University of Texas at Austin. He received his Ph.D. in Computer Science in May 2013 from Virginia Tech. One of his chief research interests is programmer productivity, which spans the spectrum from software engineering to program analysis, addressing related issues to make it easier to develop, maintain, and evolve large scale software systems.

Tiago Massoni is professor, working for the Computing and Systems Department at the Federal University of Campina Grande. His research interests include software design and verification. In addition to his academic posts he also worked as a developer at IBM in California. He holds a Doctoral degree in Computer Science from the Federal University of Pernambuco, and is a member of the ACM.

Patrícia D. L. Machado is a Professor in the Computing and Systems Department at Federal University of Campina Grande (UFCG), Brazil, since 1995. She received her PhD Degree in Computer Science from the University of Edinburgh, UK, in 2001, Master Degree in Computer Science from the Federal University of Pernambuco, Brazil, in 1994 and Bachelor Degree in Computer Science from the Federal University of Paraiba, Brazil, in 1992. Her research interests include software testing, formal methods, mobile computing, component based software development and model-driven development. Since 1998, she has produced a number of contributions in the area of software testing, including research projects, publications, tools, supervising, national/international cooperation and teaching.

Miryung Kim is an associate professor in the Department of Computer Science at the University of California, Los Angeles. Her research focuses on software engineering, specifically on software evolution. She received her B.S. in Computer Science from Korea Advanced Institute of Science and Technology in 2001 and her M.S. and Ph.D. in Computer Science and Engineering from the University of Washington under the supervision of Dr. David Notkin in 2003 and 2008 respectively. She received an NSF CAREER award in 2011, a Microsoft Software Engineering Innovation Foundation Award in 2011, an IBM Jazz Innovation Award in 2009, a Google Faculty Research Award in 2014, and an Okawa Foundation Research Grant Award in 2015. Between January 2009 and August 2014, she was an assistant professor in the Department of Electrical and Computer Engineering at the University of Texas at Austin.