An Automated Approach to Extracting Local Variables

Xiaye Chi
Beijing Institute of Technology
Beijing, China
chixiaye@icloud.com

Hui Liu*
Beijing Institute of Technology
Beijing, China
liuhui08@bit.edu.cn

Guangjie Li
National Innovation Institute of Defense Technology
Beijing, China

Weixiao Wang
Beijing Institute of Technology
Beijing, China
1120191931@bit.edu.cn

Yunni Xia
Chongqing University
Chongqing, China
xiayunni@hotmail.com

Yanjie Jiang
Beijing Institute of Technology
Beijing, China
yanjiejiang@bit.edu.cn

Yuxia Zhang
Beijing Institute of Technology
Beijing, China
yuxiazh@bit.edu.cn

Weixing Ji
Beijing Institute of Technology
Beijing, China
jwx@bit.edu.cn

ABSTRACT

`Extract local variable` is a well-known and widely used refactoring. It is frequently employed to replace one or more occurrences of a complex expression with simple accesses to a newly added variable. Although most IDEs provide tool support for extract local variables, such tools without deep analysis of the refactorings may result in semantic errors. To this end, in this paper, we propose a novel and more reliable approach, called *ValExtractor*, to conduct extract variable refactorings automatically. The major challenge of automated extract local variable refactorings is how to efficiently and accurately identify the side effect of the extracted expressions and the potential interaction between the extracted expressions and their contexts without time-consuming dynamic execution of the involved programs. To resolve this challenge, ValExtractor leverages a lightweight static source code analysis to validate the side effect of the selected expression, and to identify which occurrences of the selected expression could be extracted together without changing the semantics of the program or introducing potential new exceptions. Our evaluation results on open-source Java applications suggest that Eclipse and IntelliJ IDEA, the state-of-the-practice refactoring engines, resulted in a large number of faulty extract variable refactorings whereas ValExtractor successfully avoided all such errors. The proposed approach has been merged into (and distributed with) Eclipse to improve the safety of extract local variable refactoring.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Integrated and visual development environments**.

*Corresponding author

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616261

KEYWORDS

Software Refactoring, Extract Local Variable, Reliable, Bugs

ACM Reference Format:
Xiaye Chi, Hui Liu, Guangjie Li, Weixiao Wang, Yunni Xia, Yanjie Jiang, Yuxia Zhang, and Weixing Ji. 2023. An Automated Approach to Extracting Local Variables. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616261

1 INTRODUCTION

The term "*software refactoring*" was coined by Opdyke [29], referring to the object-oriented variant of restructuring [1]. In general, software refactoring could be defined as "*the process of changing a [object-oriented] software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*" [11]. Recently, software refactoring has been well studied as an efficient way to improve software quality [6, 20] as well as an effective way to facilitate software maintenance and evolution [21, 50]. Refactoring tools like JDeodorant [43], ReSharper [32], and built-in refactoring engines in IDEs (including Eclipse[7], IntelliJ IDEA [16], NetBeans [27], and Visual Studio [49]) have been widely used to facilitate software refactoring.

Extract local variable (or *extract variable* for short) is one of the most popular refactorings. Notably, dozens of software refactorings have been proposed, ranging from low-level refactorings like *renaming variable* to high-level refactorings like *teasing apart inheritance* [11]. By tracking the refactoring histories of programmers, Murphy-Hill et al. [25] found that *extract variable* was the second most popular software refactoring. *Extract local variable* is to create a local variable, initialize it with a selected expression, and replace one or more occurrences of the expression with direct access to the new variable. The benefits of the refactoring are twofold [10]. On one side, replacing complex expressions with a named variable may increase the readability of the program because variable names are often more readable than complex expressions. On the other side, replacing multiple occurrences of the same expression with simple

accesses to a variable may avoid repetitive computation and thus reduce code complexity.

Manually extracting local variables could be tedious, time-consuming and error-prone. It has been well known that changing source code of complex software systems could be risky [37]. The same is true for software refactoring [3]. To this end, mainstream IDEs have provided automated tool support for this refactoring. According to the survey by Golubev et al. [15], 54.7% of the surveyed developers use the IDE support to conduct*Extract* refactorings, e.g., *extract variables*. The empirical study conducted by Negara et al. [26] suggests that extract variable is frequently performed with automated tool support. Developers pferm over 80% of extract variable refactorings with automated tool support. However, such refactoring tools often simply replace all expressions that are lexically identical to the selected one without in-deep analysis on the safety of the refactoring. As a result, even with such tool support, extracting local variables could be error-prone, resulting in exceptions and semantic errors. In Section 2, we explain with motivating examples why the state-of-the-practice refactoring tools may introduce semantic errors while extracting local variables.

To this end, in this paper, we propose a novel and more reliable approach, called *ValExtractor*, to conduct extract variable refactorings automatically. The major challenge of automated extract local variable refactorings is how to efficiently and accurately identify the side effect of the extracted expressions and the potential interaction between the extracted expressions and their contexts (i.e., statements around them) without time-consuming dynamic execution of the involved programs. To resolve this challenge, ValExtractor leverages a lightweight static source code analysis to validate the side effect of the selected expression, and to identify which occurrences of the selected expression could be extracted together without changing the semantics of the program or introducing potential new exceptions. We evaluated the proposed approach on open-source applications by applying it and the baseline approaches (Eclipse and IntelliJ IDEA) to extract expressions in such applications. Our evaluation results suggested that the state-of-the-practice baselines did result in hundreds of semantic errors while conducting extract variable refactorings. Our approach, however, successfully avoided all such errors. Besides that, we also evaluated the proposed approach and Eclipse with 253 real-world extract variable refactorings discovered from 10 open-source applications. Our evaluation results suggested that Eclipse resulted in semantic errors in 19 out of the 253 cases, and another mainstream IDE *IntelliJ IDEA* resulted in semantic errors on all such 19 cases as well. In contrast, our approach succeeded in conducting all such refactorings without introducing any semantic errors.

The paper makes the following contributions:

- An automated and more reliable approach to extracting local variables in Java applications.
- A benchmark consisting of 253 real-world extract local variable refactorings.
- An evaluation of the proposed approach on the benchmark, whose replication package, including detailed instruction for replication, is publicly available [31].

```
1  private static String parseToken(String pattern, int[] indexRef) {
2    ...
3    if (pattern == null || pattern.length() > MAX_LEN) {
4      return null;
5    }
6    while (i + 1 < pattern.length()) {
7      ...
8    }
9    ...
10   char lastChar = pattern.charAt(pattern.length() - 1);
11   pattern = "Default_" + pattern;
12   int j = indexRef[0];
13   while (j < pattern.length()) {
14   ...
```

Listing 1: Motivating Example

2 MOTIVATING EXAMPLE

In this section, we explain with a motivating example the potential risks in extract variable refactorings, and how we minimize such risks. The motivating example is presented in Listing 1. Suppose that a developer realizes that there are many instances of expression "`pattern.length()`" in the motivating example (as shown in colors), and would like to replace such instances with a local variable. To this end, the developer selects the expression "`pattern.length()`" at Line 6 within *Eclipse*[1], right-clicks it, and selects menu item "`refactoring - extract local variable`" as well as the checkbox "`replace all occurrences of the selected expression with references to the local variable`". As a response to the command, Eclipse invokes JDT [8] to conduct the extract variable refactoring. The resulting source code is presented in Listing 2. Notably, conducting the same refactoring with *IntelliJ IDEA* would result in the same code.

The refactorings conducted automatically by Eclipse and IDEA are questionable. By comparing the code before and after the refactoring, we notice that Eclipse and IDEA declare a new variable (`length`) at Line 3 in Listing 2 and initializes it with the extracted expression "`pattern.length()`". It also replaces all of the four instances of the expression with the newly added variable `length` at Lines 4, 7, 11, and 14, respectively. However, the replacement is incorrect and it results in serious bugs that change the semantics of the enclosing software application. First, the newly added declaration at Line 3 is questionable. In case the input parameter `pattern` equals `null`, the declaration would result in a `null pointer exception`. In contrast, the source code before refactoring can avoid the exception because it carefully checks whether the `pattern` equals `null` (Line 3 of Listing 1) before the variable is used to access any of its properties. Second, replacing the expression "`pattern.length()`" with variable `length` at Lines 14 of Listing 2 is incorrect. The variable `pattern` has been updated at Line 12. Consequently, at Line 14 the variable `length` (initialized at Line 3) is not equivalent to the original expression "`pattern.length()`". As a result, replacing the expression with variable `length` at Line 14 is incorrect, which may result in fewer iterations at Line 14.

To avoid such errors, in this paper, we propose an automated approach *ValExtractor* to conduct extract variable refactorings. It successfully conducts the refactoring as shown in Listing 3 and avoids all bugs introduced by the state-of-the-practice IDEs (i.e.,

[1]Do not use version 4.26.0 or later where our approach has been integrated.

```
1  private static String parseToken(String pattern, int[] indexRef) {
2    ...
3    int length = pattern.length();
4    if (pattern == null || length > MAX_LEN) {
5      return null;
6    }
7    while (i + 1 < length) {
8      ...
9    }
10   ...
11   char lastChar = pattern.charAt(length - 1);
12   pattern = "Default_" + pattern;
13   int j = indexRef[0];
14   while (j < length) {
15   ...
```

Listing 2: After Refactoring(by Eclipse or IDEA)

```
1  private static String parseToken(String pattern, int[] indexRef) {
2    ...
3    if (pattern == null || pattern.length() > MAX_LEN) {
4      return null;
5    }
6    int length = pattern.length();
7    while (i + 1 < length) {
8      ...
9    }
10   ...
11   char lastChar = pattern.charAt(length - 1);
12   pattern = "Default_" + pattern;
13   int j = indexRef[0];
14   while (j < pattern.length()) {
15   ...
```

Listing 3: After Refactoring (by `ValExtractor`)

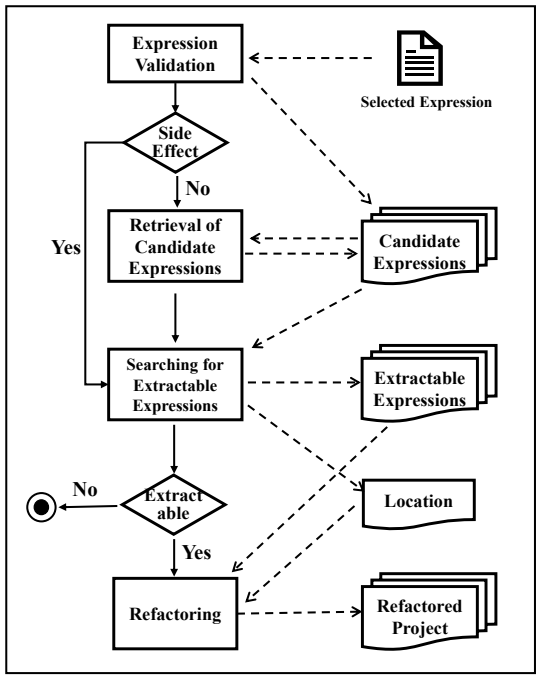

Figure 1: Overview of `ValExtractor`

Eclipse and IDEA). `ValExtractor` works as follows. First, given the selected expression "`pattern.length()`" at Line 6 in Listing 1, `ValExtractor` infers that the new variable declaration should be added between Line 5 and Line 6 in Listing 1 if the selected expression alone should be extracted as a new variable. After that, `ValExtractor` validates that the variable and the expression (to be replaced with access to the variable) are equivalent at Line 6, and the expression itself does not have any side effect. Consequently, the selected expression (at Line 6) is extractable, and it is added as an *extractable expression*.

`ValExtractor` keeps finding more *extractable expressions* that could be extracted together with the selected expression. To this end, it turns to the expression "`pattern.length()`" at Line 10 of Listing 1 because it is the closest expression to the selected expression at Line 6 and it is literally identical to the selected expression. It repeats the inference in suggesting where the new variable should be declared as well as the validation of potential side effect as introduced in the preceding paragraph. This time, `ValExtractor` suggests that the new variable could be declared between Line 5 and Line 6 in Listing 1 and that replacing both of the expressions (at Lines 6 and 10) is safe. Consequently, the expression at Line 10 is also added as an *extractable expression*.

`ValExtractor` comes to the next expression "`pattern.length()`" at Line 13 of Listing 1. While validating the side effect of the expressions between Line 10 and Line 13, ValExtractor finds that the statement at Line 11 has side effect on the selected expression (i.e., it may change the value of the expression "`pattern.length()`"). As a result, executing the same expression appearing before and after Line 11 may result in different values, and thus we cannot extract the expressions at Line 6 (before Line 11) and Line 13 (after Line 11) together. To this end, ValExtractor discards the expression at Line 13 as well as other expressions beyond it.

Finally, it reverses the searching direction, and turns to the expression "`pattern.length()`" at Line 3. It infers that the new variable should be declared and initialized before Line 3 in Listing 1. However, the initialization of the new variable with the expression "`pattern.length()`" before Line 3 may result in a `null pointer exception` (when `pattern` equals `null`) that may not happen before the refactoring. Consequently, ValExtractor discards this expression as well as other expressions before it (if any).

As a result of the preceding static analysis, `ValExtractor` extracts two *extractable expressions* at Lines 6 and 10 of Listing 1, avoiding all bugs introduced by Eclipse JDT.

3 APPROACH

3.1 Overview

An overview of the proposed approach (`ValExtractor`) is presented in Fig. 1. It takes as input a selected expression and its enclosing project. With such input, ValExtractor validates whether the selected expression has side effect and validates iteratively whether other literally identical expressions within the same method could be extracted together. Overall, ValExtractor works as follows:

- Expression validation: It adds the selected expression as a *candidate expression*, and validates whether the selected expression has side effect. If yes, it skips the next step.

- Retrieval of candidate expressions: It retrieves all expressions within the enclosing method that are literally identical to the selected expression, taking them as *candidate expressions*.
- Search for extractable expressions: It takes a greedy strategy to search for candidate expressions that could be extracted together with the selected one (called *extractable expressions*), and suggests where the new variable should be declared. If none of the candidate expressions could be extracted, `ValExtractor` terminates and no refactoring would be conducted. Otherwise, `ValExtractor` turns to the next step.
- Refactoring: Finally, ValExtractor conducts extract variable refactoring by declaring and initializing a new variable and replacing all of the extractable expressions with accesses to the variable.

Details of the key steps are presented in the following sections, and the full list of preconditions when a set of literally identical expressions could be extracted by an extract variable refactoring is presented as an online appendix [12].

3.2 Expression Validation

The validation of the selected expression is composed of two parts. The first part validates whether the selected expression is suitable for extraction. Not all expressions could be extracted as local variables. For example, "`this.length`" in assignment "`this.length=5`", "`ArrayList<String>()`" in statement "`list= new ArrayList<String>()`", and "`id.isEmpty()`" in statement "`st.id.isEmpty()`" cannot be extracted as variables. ValExtractor terminates (i.e., refuses to conduct the refactoring) if the selected expression is one of the following expressions: parameters, left values, declarations, single null literal, expressions in annotations, incomplete expressions, void expressions, enumeration expressions in switch cases, expressions used in initializer or updater of *for* statements, name properties, and expressions outside methods.

In the second part, ValExtractor validates whether the selected expression has side effect. An expression has `a side effect` if executing the same expression (one or more times) is not semantically equivalent to a single execution of the expression. For example, the expression `stack.pop()` has side effect because repeating it `n` times may remove additional `n` elements from the stack. Consequently, the following code

```
Print(stack.pop());
Print(stack.pop());
```

is not equivalent to the following code:

```
value=stack.pop()
Print(value);
Print(value);
```

If the selected expression has side effect, we cannot extract it together with other expressions that are literally identical to it. In this case, the selected expression is taken as the only *candidate expression*, i.e., ValExtractor will extract no more than one expression.

ValExtractor validates the side effect of the selected expression by checking whether the expression has updated states of the system, generated outputs, or consumed system inputs. An expression has updated the states of the system if and only if the expression (including methods called directly or indirectly by it) has updated any software entities whose lifetime is beyond the execution of the selected expression. ValExtractor identifies generation of outputs and consumption of system inputs by comparing the executed statements against a list of manually marked Java input/output APIs. If any of the marked APIs is executed directly or indirectly by the selected expression, it has side effect.

```
1  InstantConverter conv =
       ConverterManager.getInstance() .getInstantConverter (lhsObj);
2  Chronology lhsChrono = conv.getChronology(lhsObj, (Chronology) null);
3  long lhsMillis = conv.getInstantMillis (lhsObj, lhsChrono);
4  conv = ConverterManager.getInstance() .getInstantConverter (rhsObj);
5
6  public static ConverterManager getInstance() {
7    if (INSTANCE == null) {
8      INSTANCE = new ConverterManager();
9    }
10   return INSTANCE;
11 }
```

Listing 4: Expressions Updating Empty Fields Only

An exception to the preceding rules is that we allow the selected expression to initialize fields that are initially `null`. Listing 4 presents a typical example of such initialization. The selected expression "`ConverterManager.getInstance()`" is to retrieve the static field `manager` of class `ConverterManager`. However, if the variable equals `null` (i.e., it has not yet been initialized), the expression would initialize it with a brand new object (Line 8). Consequently, although the selected expression has the possibility to update the field `manager`, repeating the expression multiple times is semantically equivalent to a single execution of the same expression.

3.3 Retrieval of Candidate Expressions

First, ValExtractor automatically infers the scope of the selected expression, noted as *ScopeExp*. The scope of the expression specifies where the expression is syntactically accessible. Consequently, the scope is the intersection of the scopes of all elements involved in the expression. For example, the scope of the expression "`list.add(item)`" is the intersection of the scope of the variable "`list`" and the scope of the parameter "`item`".

To retrieve candidate expressions that may be extracted together with the selected expression, ValExtractor searches for all expressions within *ScopeExp* that are lexically identical to the selected expression. For each of the retrieved expressions, ValExtractor also validates whether it is suitable for extraction in the same way as it validates the selected expression in Section 3.2. All expressions passing the validation are added as *candidate expressions*.

3.4 Searching for Extractable Expressions

For a selected to-be-extracted expression and a sequence of candidate expressions, ValExtractor searches for extractable expressions by Algorithm 1. The output of the algorithm is a suggested location for declaring and initializing the new variable (that should be introduced by extract variable refactoring), and a sequence of extractable expressions that could be replaced with accesses to the new variable.

3.4.1 Iteration. On each iteration (Lines 5-16 in Algorithm 1), ValExtractor validates whether an additional candidate expression

Algorithm 1: : Searching for Extractable Expressions

```
   Input: selExp ; // selected expression
          canExps // candidate expressions
   Output: extExps ; // extractable expressions
          Loca // location for variable declaration
1  searchDir = 2
2  extExps = ∅
3  Loca = −1
4  cExp = selExp
5  while cExp != null && searchDir > 0 do
6      exps = {cExp} ∪ extExps
7      canExps.remove(cExp)
8      Loc = InferBestLoc(exps)
9      if ChangeExpValue(Loc, exps, selExp) OR
         AdditionalException(Loc, exps) then
10         searchDir = searchDir - 1
11     else
12         extExps = exps
13         Loca = Loc
14     end
15     cExp = getNext(canExps, extExps, searchDir, selExp)
16 end
```

Algorithm 2: : Auxiliary Functions

```
1  Function InferBestLoc(exps):
2      Parent = CommonParentNode(exps)
3      if Parent instanceof Block then
4          nodes = Parent.getChildren();
5          for (i = 0; i < nodes.size; i++) do
6              if nodes[i].contain(exps) then
7                  Loc = StmtBasedLoc(nodes[i].Loc)
8                  break
9              end
10         end
11     else
12         Loc = StmtBasedLoc(Parent.Loc)
13     end
14 return Loc
15 Function ChangeExpValue(Loc, exps, selExp):
16     vars_r_exp = VariablesReadby(selExp)
17     sts = StatementsWithinScope(Loc, exps)
18     for each statement in sts do
19         vars_w_st = VariablesWrittenby(statement)
20         if vars_w_st ∩ vars_r_exp != ∅ then
21             return true
22         end
23     end
24 return false
25 Function AdditionalException(Loc, exps):
26     for each exp in exps do
27         Ncds = NullCheckConditions(loc, exp)
28         Nins = Ncds.CheckedInstances ∩
             exp.AccessedInstances
29         Tcds = TypeCheckConditions(loc, exp)
30         Tins = Tcds.CheckedInstances ∩
             exp.TypeCastedInstances
31         if Nins ≠ ∅ OR Tins ≠ ∅ then
32             return true
33         end
34     end
35 return false
```

cExp could be extracted together with other extractable expressions in *extExps*. For convenience, we define a new set *exps* = {*cExp*} ∪ *extExps* at Line 6. The validation is composed of two steps. On the first step (Line 8), ValExtractor infers the best location for declaring and initializing the new variables in case all expressions in *exps* could be extracted together as a single variable. On the second step (Line 9), with methods *ChangeExpValue* and *AdditionalException*, ValExtractor validates:

(1) Whether any statements between *Loc* (where the new variable would be initialized) and the expressions in *exps* (where the variable would be accessed) would change the value of the selected expression, i.e., would update any variables that are read by the expression. Notably, direct and indirect method invocations are analyzed as well.
(2) Whether the refactoring could result in additional exceptions that may not be raised before the refactoring.

If either of the validation fails (i.e., returning `true`), the candidate expression cannot be extracted together with the selected expression. Otherwise, the current expression *cExp* is added as an extractable expression (Lines 6 and 12). Methods *InferBestLoc*, *ChangeExpValue*, and *AdditionalException* are explained in details in Section 3.4.2 and Section 3.4.3.

Notably, the algorithm validates the selected expression itself on the first iteration (Line 4 and Lines 5-14). If the selected expression passes the validation, the algorithm would leverage the method `getNext` (Line 15) to retrieve the next candidate expression in *canExps* that may be extracted together with the selected expression. On each iteration, the algorithm may add an additional candidate expression to the extractable expressions (Lines 6 and 12). The algorithm first searches for extractable expressions by scanning statements following the selected expression (*selExp*). After that, it reverts the searching direction and scans statements before the selected expression. It leverages the flag "`searchDir`" to control the searching direction. If "`searchDir==2`", `getNext` would return the candidate expression in *canExps* that is 1) not included in *extExps* and 2) appears in the source code file after all expressions in *extExps* but before other expressions in *canExps* (candidate expressions). If "`searchDir==1`", `getNext` would reverse the search direction: returning the candidate expression that appears in the source code file before all expressions in *extExps* but after other expressions in *canExps*. When "`searchDir==0`" is true, the algorithm stops searching for more extractable expressions.

3.4.2 Inferring the Best Location for Variable Declaration. The function *InferBestLoc* in Algorithm 2 shows how to infer the best location for variable declaration. ValExtractor leverages the AST of the enclosing method to find the lowest common parent node for all of the expressions in *exps* (Line 2). If the parent node is not a *block* [2], ValExtractor suggests inserting the variable declaration immediately before the parent node (Line 12). Otherwise, ValExtractor suggests inserting the declaration as a child node of the parent node (block) and this child node should be defined just before the first child node of the block that contains any expressions in *exps* (Lines 4-10). Notably, we leverage the method `StmtBasedLoc` (Lines 7 and 12) to avoid inserting the variable declaration within an existing statement. It automatically returns the first place where a variable declaration could be inserted before the given absolute location.

3.4.3 Checking Preconditions. ValExtractor employs two functions *ChangeExpValue* and *AdditionalException* in Algorithm 2 to check preconditions for the refactoring. Function *ChangeExpValue* validates whether any statement between *Loc* (where the new variable would be initialized) and the expressions in *exps* (where the variable would be accessed) would change the value of the selected expression. If yes, extracting all expressions in *exps* together could result in semantic errors, and thus the current expression *cExp* is discarded for safety (Lines 9-10 in Algorithm 1). *ChangeExpValue* identifies all variables that are accessed by the selected expression (Line 16). All statements between *Loc* and the expressions in *exps* are retrieved and noted as *sts* (Line 17). For each of the statements in *sts*, ValExtractor validates whether it updates any element that has been read by the selected expression (Lines 20-22). If yes, *ChangeExpValue* terminates the validation and returns `true` (i.e., the precondition is not satisfied).

Besides the variable access-based precondition checking, ValExtractor also leverages function *AdditionalException* to conduct an exception-based precondition checking (Lines 25-35). It is likely that the execution of the same expression in some places would result in exceptions whereas the execution in other places would not result in such exceptions. A typical example is presented in Listing 2. The same expression "`pattern.length()`" at Line 3 may result in a *null pointer exception* whereas the same expression at Line 4 (before it is replaced by variable `length`) will not. The latter avoids the null pointer exception because of the preceding condition ("`pattern==null`"): if `pattern` equals `null`, the expression at Line 4 would not be executed, which avoids the null pointer exception. Consequently, extracting the expression at Line 4 as a new variable (at Line 2) as presented in Listing 2 may result in additional exceptions, which is called exception-based side effect. Of the current version, we focus on null pointer exceptions and class cast exceptions only, and thus we only check 1) whether any instances accessed by the selected expression have been checked against *null* (Lines 27, 28, 31), and 2) whether any instances typecasted by the selected expression have been checked against the given types (Lines 29, 30, 31).

3.5 Refactoring

Suppose that ValExtractor identifies a sequence of extractable expressions (*extExps*) and suggests declaring the new variable on location *Loca*, ValExtractor would conduct the suggested extract variable refactoring as follows.

```
1  // Before Refactoring
2  for (Good g: goodlist )
3      if ( g.getPrice() > 0)
4          sum += g.getPrice() * discount ;
5  ------------------------------------
6  // After Extract Variable Refactoring
7  for (Good g: goodlist ) {
8      double price = g.getPrice();
9      if ( price > 0)
10         sum += price * discount ;
11 }
```

Listing 5: Inserting into Single-Statement `For-Iteration`

- First, ValExtractor infers the data type of the new variable by exploiting `TypeBinding` of `JDT` [17].
- Second, ValExtractor inserts the declaration of the new variable and initializes it with the selected expression on location *Loca*. If it is inserted into a single-statement complex structural node, like `if-else-statement`, `while-iteration`, `lambda-statement`, and `for-iteration`, ValExtractor would insert "`{`" and "`}`" to mark the whole code block. A typical example is presented in Listing 5. Without the newly inserted "`{`" at Line 7 and "`}`" at Line 11, the `if-statement` (Lines 9-10) would be moved out of the `for-statement`, which would result in semantic errors because the if-statement constitutes the body of the `for-statement` in the original code.
- Finally, ValExtractor replaces all extractable expressions in *extExps* with direct access to the new variable. Notebly, we leverages the API provided by JDT to recommend variable name based on the selected expression and its expected type.

4 EVALUATION

4.1 Research Questions

The evaluation investigates the following research questions:

- **RQ1:** Does ValExtractor improve the state of the practice in automating extract variable refactorings?
- **RQ2:** Why do Eclipse and IDEA result in faulty extract variable refactorings and how does ValExtractor avoid such errors?
- **RQ3:** Does ValExtractor outperform the state of the art in precondition checking for extract local variable refactorings?

RQ1 concerns the comparison among ValExtractor, Eclipse, and IntelliJ IDEA. Eclipse (version 4.23.0 released in 2022) and IDEA (2022-03 release) were selected for comparison because they represent the state of the practice, and they are widely used in the industry. Notably, we did not employ the latest version of Eclipse because the proposed approach has been integrated into Eclipse since version 4.26.0. RQ2 concerns the reasons/mechanism for introducing/avoiding errors in automated extract variable refactorings. RQ3 concerns the precondition checking of extract local variable refactorings. To answer RQ3, we compare ValExtractor against JRRT [35] that represents the state of the art in precondition checking for software refactorings. Notably, while investigating RQ1, we do not compare ValExtractor against JRRT because JRRT has its special strategy: It always extracts the selected expression only (not with any other lexically identical expressions) [35]. It is completely

different from ValExtractor, Eclipse and IDEA, making it hard to compare them directly concerning the overall performance.

4.2 Subject Applications

We leveraged all applications in the well-known bug repository Defects4J [5](version 1.1.0) for the evaluation. In total, Defects4J contains 6 Java applications, as specified in the first column of Table 1. The size (LOC) of the applications varied from 61,298 to 230,135. We reused such applications because we were familiar with this repository, which might facilitate manual validation/analysis of the evaluation results. Besides that, the applications are widely used open-source applications from different domains, which might help reduce potential bias and facilitate replication of the evaluation.

4.3 Process

On each subject application, the evaluation was conducted as follows. First, we retrieved all expressions that appeared multiple times within the same method. Each of the retrieved expressions could be represented $exp_i =< txt, Locs >$ where txt is the text of the expression whereas $Locs$ is a sequence of locations where the expression appears. Notably, expressions (e.g., expression "*a.b*" in statement "*a.b=4*") that Eclipse JDT refuses to extract had been excluded. We leveraged the APIs of JDT to automate the exclusion. Second, for each expression exp_i, we randomly selected one of its occurrences as the *selected expression* and fed this selected expression to the evaluated approaches (ValExtractor, Eclipse, and IDEA) to conduct extract variable refactoring. If the approaches generated different outputs, we marked the refactorings as a triple of *inconsistent refactorings*. Third, from the resulting triples of inconsistent refactorings, we randomly sampled n triples for manual checking. The size of the sample (i.e., n) guarantees an error margin of 5% and a confidence level of 95% [34]. We also sampled m triples of consistent refactorings in the same way. Finally, three highly experienced developers with over three years of Java expertise manually checked the sampled triples of inconsistent/consistent refactorings. They discussed together and classified the refactorings as one of the followings:

- `Buggy`: The refactoring was incorrect because it introduced semantic errors.
- `Correct`: The refactoring was conducted correctly.
- `Imperfect`: The refactoring did not introduce any semantic errors, but it missed some extractable expressions.

4.4 RQ1: Improving the State of the Practice

The evaluation results are presented in Table 1. `#Cases` in Table 1 specifies how many cases have been used for the evaluation, i.e., how many times the proposed approach (and the baseline approaches) have been applied to extract variables. `#Consistent` specifies how many times the evaluated approaches resulted in identical results. `#Inconsistent` specifies how many times they resulted in different (inconsistent) refactorings.

From Table 1, we make the following observations:

- First, more than 11 thousand cases have been involved in the evaluation. Such a large number of cases enables a thorough evaluation of the approaches.

Table 1: Evaluation Results

Applications	#Cases	#Consistent	#Inconsistent
Closure	2,465	1,957	508
Jfreechart	3,831	3,292	539
Joda	510	335	175
Lang	565	388	177
Math	3,516	3,084	432
Mockito	135	118	17
`Total`	`11,022`	`9,174`	`1,848`

- Second, the evaluated approaches frequently resulted in inconsistent refactorings. The inconsistent cases account for 16.8%=1,848/11,022 of the evolved cases. Such a non-trivial ratio of inconsistent cases may suggest that extract variables automatically is error-prone, which further motivates the research presented in this paper.
- Third, on all of the six subjection applications, the three approaches reported inconsistent cases.

According to the process introduced in Section 4.3, we randomly sampled 318 inconsistent cases and 369 consistent cases, and requested three developers to manually check the samples. The evaluation results suggested that all refactorings on the 369 consistent cases were correct. The manual checking results on inconsistent cases are presented in Table 2. The columns "`#correct`", "`#imperfect`" and "`#buggy`" present the numbers of correct, imperfect, and buggy extract variable refactorings conducted by the evaluated approaches, respectively. From this table, we make the following observations:

- Eclipse resulted in a large number of buggy extract variable refactorings. On 243 out of the 318 inconsistent cases, Eclipse resulted in buggy refactorings. It resulted in errors on all of the evaluated applications.
- IDEA resulted in the largest number (263) of faulty refactorings in the sampled 318 inconsistent cases. IDEA and Eclipse frequently failed on the same cases. On 233 out of the 318 cases, both IDEA and Eclipse resulted in errors.
- ValExtractor avoided all of the bugs introduced by Eclipse or IDEA. It did not result in any faulty refactorings.
- All of the evaluated approaches resulted in imperfect extract variable refactorings, i.e., missing some extractable expressions. The major reason for such imperfect refactorings is that they employ conservative tactics to avoid errors, and thus any *potentially* unsafe expressions would be ignored. We also notice that ValExtractor reported more imperfect refactorings than Eclipse and IDEA because it pays more attention to safety and thus its tactic is more conservative.

We also assessed the efficiency of ValExtractor. On a personal computer with 8 GB memory and Intel i7-8550U CPU, 91.6% of the refactorings were finished by ValExtractor within 2 seconds, 96.5% within 5 seconds, and 98.3% within 10 seconds. The median execution time for a single refactoring was 0.20 seconds, comparable to that (0.13 seconds) of Eclipse. The maximal execution time (390 seconds) is substantially larger than that (1 second) of Eclipse. ValExtractor is more time-consuming because it leverages more complex code analysis to detect potential errors. To reduce the maximal time, we may present a maximal time slot. When ValExtractor

Table 2: Inconsistent Refactorings

Applications	ValExtractor			Eclipse			Intellij IDEA		
	#Correct	#Imperfect	#Buggy	#Correct	#Imperfect	#Buggy	#Correct	#Imperfect	#Buggy
Closure	83	4	0	23	1	63	14	0	73
Jfreechart	93	4	0	28	1	68	25	0	72
Joda	29	1	0	3	0	27	2	0	28
Lang	25	3	0	8	2	18	6	1	21
Math	70	2	0	8	1	63	7	0	65
Mockito	4	0	0	0	0	4	0	0	4
Total	304	14	0	70	5	243	54	1	263

Table 3: Reasons for Buggy Refactorings

Applications	Exception in Initialization	Changed Values	Side Effect
Closure	18(23.7%)	30(39.5%)	28(36.8%)
Jfreechart	5(6.5%)	29(37.7%)	43(55.8%)
Joda	5(18.5%)	14(51.9%)	8(29.6%)
Lang	7(29.2%)	10(41.7%)	7(29.2%)
Math	9(13.8%)	33(50.8%)	23(35.4%)
Mockito	2(50.0%)	2(50.0%)	0(0.0%)
Total	46(16.8%)	118(43.2%)	109(39.9%)

```
1  public void predict(final RealVector u)
2          throws DimensionMismatchException {
3      ...
4      int var_948 = u.getDimension();
5      if (u != null &&
6      u.getDimension() var_948 != controlMatrix.getColumnDimension()) {
7          throw new DimensionMismatchException(u.getDimension() var_948,
8                                               controlMatrix.getColumnDimension());
9      }
10 }
```

Listing 6: Faulty Refactoring (Exceptions in Initialization)

runs out of the slot, it stops searching for additional extractable expressions, and extracts all extractable expressions at hand. Note that such a strategy does not increase the risk of faulty refactorings.

We conclude that extract variable refactoring is error-prone even with the state-of-the-practice refactoring tools. ValExtractor is much more reliable than the widely used Eclipse and IDEA.

4.5 RQ2: Reasons for Faulty Refactorings

According to the errors introduced by such refactorings, We collected faulty extract variable refactorings conducted by IDEA and/or Eclipse, resulting in a total of 273 cases, and classified them into three categories. The results of the classification are presented in Table 3. On this table, the numbers outside parenthesis are the absolute numbers of buggy refactorings falling into given categories. The percentage in parenthesis equals to the absolute number divided by the total number of buggy refactorings on the given application.

The category "`exception in initialization`" refers to such refactorings where the initialization of the new variables may result in exceptions that may not be raised before the refactorings. A typical example is presented in Listing 6. The example code comes from the open-source project `Math`. Eclipse extracted the expression "*u.getDimension()*" at Line 6 and Line 7 as a new variable and initialized it at Line 4. However, the initialization might raise null pointer exceptions when `u` equals `null`. In contrast, even if `u` equals `null`, the source code before refactoring can avoid the exception because

```
1  private RuleSet getLastRuleSet() {
2      int var_656 = iRuleSets.size();
3      if (iRuleSets.size() var_656 == 0) {
4          addCutover(Integer.MIN_VALUE, ... 0);
5      }
6      return iRuleSets.get(iRuleSets.size() var_656 - 1);
7  }
8  public DateTimeZoneBuilder addCutover(int year, ... , int millisOfDay){
9      if ( iRuleSets.size() > 0) {
10         ...
11     }
12     iRuleSets.add(new RuleSet());
13     return this;
14 }
```

Listing 7: Faulty Refactoring (Changed Values)

it leverages the condition "`u!=null`" at Line 5 to avoid the execution of the expression "*u.getDimension()*" in case `u` equals `null`. ValExtractor avoided this error by extracting only the second occurrence of the expression (Line 7) and initializing the new variable within the body of the *if statement*. As shown in Table 3, 16.8% of the faulty extract variable refactorings conducted by Eclipse/IDEA fall into this category.

The second category, "`changed values`", refers to extract variable refactorings where the expressions replaced with the same variable have different values although they are literally identical. Listing 7 presents a real-world example conducted by Eclipse and Idea on project `Joda`. Eclipse extracted the expression "*iRuleSets.size()*" at Line 3 and 6 as a new variable `var_656` and initialized the new variable at Line 2. However, the method invocation at Line 4 would change the set `iRuleSets`, and thus would change the size of the set. As a result, at Line 6, the original expression "*iRuleSets.size()*" is not equal to the value of `var_656` that is initialized at Line 2. As a result, replacing the expression with the variable at Line 6 as Eclipse did result in semantic errors. ValExtractor avoided this error by replacing the first expression at Line 3 only. The preconditions in Section 3.4.3 successfully prevented ValExtractor from extracting the expression at Line 6 together with the selected one (Line 3). Notably, such kind of errors is hard to find because they do not raise any exceptions or warning. However, as suggested in Table 3, around half the faulty extract variable refactorings conducted by Eclipse and IDEA would result in such kind of hard-to-detect errors. It may suggest how dangerous extract variable refactorings could be.

The last category, "`side effect`", refers to such extract variable refactorings where execution of the selected expressions has side effect, i.e., executing multiple times the same expression is not semantically equivalent to a single execution of the same expression.

```
private String format(JSError error, boolean warning) {
  SourceExcerptProvider source = getSource();
  int var_333 = error.lineNumber;
  String sourceExcerpt = source == null ? null :
      excerpt.get(source, error.sourceName,
      error.lineNumber var_333, excerptFormatter);
  ...
}
```

Listing 8: Safe Refactoring Rejected by JRRT

For space limitation, we present a typical example of this category online [28]. Note that the selected expression in this example is a method invocation *parseEscapeChar()* that retrieves a char from field *pattern*. It has side effect because the method returns different char on each invocation: The method moves a pointer on each invocation so that each char in *pattern* is read once and for all. ValExtractor identifies the side effect with the *validation of selected expressions* as introduced in Section 3.2.

we conclude that extract variable refactorings may result in various categories of errors, and ValExtractor avoided them.

4.6 RQ3: Comparing against JRRT

To compare ValExtractor against JRRT concerning their performance in precondition checking, we applied both ValExtractor and JRRT to the 11,022 expressions that had been randomly selected in Section 4.3. Their checking results were inconsistent if and only if one agreed to extract the selected expression whereas the other rejected it. In total, for 2,273 out of the 11,022 expressions, they led to inconsistent results. We randomly sampled 329 inconsistent cases and 368 consistent cases for manual checking. The sizes of the samples guaranteed an error margin of 5% and a confidence level of 95% [34]. We requested the same participants in Section 4.3 to manually validate whether the selected expressions could be extracted (as local variables) safely. The evaluation results suggested that both of them were correct in all of the 368 consistent cases. However, JRRT made incorrect decisions on all of the 329 inconsistent cases.

On 317 out of the 329 inconsistent cases, JRRT suggested not to refactor whereas ValExtractor suggested that the refactorings were safe. Manual checking suggested that all of the 317 refactorings suggested by ValExtractor (but forbidden by JRRT) were safe and correct. A typical example is presented in Listing 8. JRRT suggested not to refactor because the refactoring may "*unlock dependencies*" that means JRRT mistakenly assumes that the refactoring may break existing dependencies in program and potentially result in errors or unpredictable behavior. However, the extract refactoring (in Listing 8) suggested by ValExtractor is safe and correct.

On 12 out of the 329 inconsistent cases, JRRT suggested refactoring whereas ValExtractor suggested not refactoring. Manual checking confirmed that all of the 12 refactoirngs suggested by JRRT were faulty. A typical example is presented in Listing 9. The refactoring suggested by JRRT as shown in Listing 9 may result in null pointer exception if *info* equals *null*. ValExtractor rejected the refactoring because it realized the potential exception caused by the refactoring.

```
public ImmutableList<String> getTemplateTypeNames() {
    ImmutableList<String> var_2522 = info.templateTypeNames;
    if (info == null || info.templateTypeNames var_2522 == null) {
      return ImmutableList.of();
    }
    return info.templateTypeNames;
}
```

Listing 9: Faulty Refactoring Suggested by JRRT

5 CASE STUDY

In the preceding section, we evaluated ValExtractor by randomly applying it to expressions in real-world open-source applications. However, such randomly selected refactorings could be different from what developers did in the industry. To this end, in this section, we further evaluated ValExtractor with real-world extract variable refactorings that had been actually conducted by developers on open-source applications.

5.1 Subject Applications

To collect a large number of real-world extract variable refactorings, we should select such applications with rich evolution histories. To this end, we selected subject applications collected by the well-known bug repository *GrowingBugs* [18]. *GrowingBugs* contains more than one thousand bugs automatically discovered from the evolution histories of open-source applications. In total, it involves 80 well-known open-source applications from different domains, and such applications are developed and maintained by different teams. All of the applications contain rich evolution histories, which makes them suitable for discovering extract variable refactorings.

5.2 Process

To discover extract variable refactorings that have been conducted by the original developers, we applied `RefactoringMiner` [46] to the evolution histories of the selected applications. RefactoringMiner was selected because it represents the state of the art in discovering refactoring activities. Notably, we ignored all refactorings except for extract local variable refactorings. Besides, we also ignored the following extract local variable refactorings:

- Extract variable refactorings where the new variables are not initialized on their declaration. The proposed approach (and the baseline approach) are doomed to initialize the new variables introduced by extract variable refactorings, which may conflict with the discovered refactorings.
- Extract variable refactorings where the selected expressions are composed of constants only. In this case, ValExtractor is always equivalent to the baseline approaches, and thus there is no need to compare them based on such refactorings.

For each of the collected refactorings, we tried to re-apply the refactoring with the evaluated approaches independently. Notably, RefactoringMiner cannot tell us which expression had been selected when the discovered refactoring was conducted manually by the developers. To this end, we randomly selected one of the extracted expressions to invoke the evaluated approaches. In case a refactoring conducted by the evaluated approaches was identical to the discovered one, the refactoring was classified as `#correct`. Otherwise, we requested three developers to manually check the refactoring (noted

Table 4: Comparison against Real-World Extract Variable Refactorings

Applications	ValExtractor			Eclipse			Intellij IDEA		
	#Correct	#Imperfect	#Buggy	#Correct	#Imperfect	#Buggy	#Correct	#Imperfect	#Buggy
Codec	12	0	0	10	0	2	10	0	2
Compress	37	0	0	35	0	2	35	0	2
Jackrabbit	54	0	0	53	0	1	53	0	1
Joda	4	0	0	2	0	2	2	0	2
Johnzon	7	0	0	6	0	1	6	0	1
Jsoup	4	0	0	3	0	1	3	0	1
Math	72	0	0	71	0	1	71	0	1
Nifi	32	0	0	27	0	5	27	0	5
Storm	8	0	0	7	0	1	7	0	1
Tinkerpop	23	0	0	21	0	2	21	0	2
Total	253	0	0	234	0	19	234	0	19

```
1  public void luDecompose() throws InvalidMatrixException {
2   ...
3  for (int row = 0; row < col; row++) {
4      BigDecimal[] luRow = lu[row];
5      sum = luRow~~lu[row]~~[col];
6      for (int i = 0; i < row; i++)
7          sum = sum.subtract(luRow~~lu[row]~~[i].multiply(lu[i][col]));
8      luRow~~lu[row]~~[col] = sum;
9  }
10  ...
11 for (int row = col; row < nRows; row++) {
12     BigDecimal[] luRow = lu[row];
13     sum = lu[row][col];
14     for (int i = 0; i < col; i++)
15         sum = sum.subtract( luRow~~lu[row]~~[i].multiply(lu[i][col]));
16     luRow~~lu[row]~~[col] = sum;
17     ...
18 }
```

Listing 10: Incorrect Ground Truth

as *potential faulty refactoring*) against the ground truth (i.e., the discovered refactoring). The manual checking may result in three different conclusions: *Buggy refactorings*, *imperfect refactorings*, or *incorrect ground truth*. In case of incorrect ground truth (generated automatically by RefactoringMiner), the developers manually fixed the ground truth and re-evaluated the approaches with the fixed ground truth. In total, we found and fixed 85 incorrect ground truth. Listing 10 is an example of incorrect ground truth. The extract variable refactoring at Lines 4-8 is independent of that at Lines 12-16 although they are highly similar. However, `RefactoringMiner` reported the two independent refactorings as a single refactoring by mistake.

The three participants had rich experience in software development and software refactoring. They collaborated together and discussed to reach agreements on all of the cases. To minimize the cost of the manual checking, we only checked the top 10 applications with the most potential faulty refactorings (i.e., refactorings that were different from the ground truth).

5.3 Results

In total, 253 real-world extract variable refactorings were employed for the evaluation whose results are presented in Table 4. From the table, we make a series of observations.

First, Eclipse (and IDEA) resulted in semantic errors on 19 out of the 253 cases, with an error rate of 7.5%=19/253. Note that Eclipse and IDEA failed on exactly the same cases, and thus in Table 4 Columns 5-7 are exactly the same as Columns 8-10. We also observe that they resulted in errors on all of the 10 subject applications.

Second, we observe that ValExtractor did not result in any semantic errors. It succeeded in all of the 253 cases without missing any extractable expressions or introducing any semantic errors. It may suggest that ValExtractor is much more reliable than Eclipse and IDEA in extracting variables.

Finally, we observe from the table that none of the evaluated approaches resulted in any imperfect extract variable refactorings where some extractable expressions were missed. It is different from what we found in the random evaluation in Section 4 where all of the evaluated approaches resulted in dozens of imperfect refactorings. One possible reason is that refactorings conducted by developers in the industry could be essentially different from randomly constructed refactorings. Another possible reason is that the number of refactorings involved in this case study is substantially smaller than that involved in Section 4. Besides that, in practice, developers have unit tests that may prevent faulty refactorings from being committed to repositories.

Based on the preceding analysis, we conclude that developers have a non-negligible possibility to introduce semantic errors in conducting extract variable refactorings even equipped with the state-of-the-practice refactoring tools. The proposed approach, however, can successfully avoid such errors.

5.4 Threats to Validity

A threat to external validity is that only a limited number (253) of real-world refactorings were used for the case study. Because building the benchmark involved intensive human intervention, it is hard to enlarge the benchmark. To reduce the threat, we also leveraged more than ten thousand randomly sampled refactoring opportunities in Section 4 for the evaluation.

A threat to construct validity is that the requested manual checking could be inaccurate. In both the evaluation in Section 4 and the case study in Section 5, manual checking served as the ground truth for the evaluation. However, manual checking could be inaccurate, which might in turn result in an inaccurate measure of the performance of the evaluated approaches. To reduce the threat, we requested multiple participants to check the refactorings, and they reached agreements on all of the cases.

6 DISCUSSIONS

It remains challenging to determine the side effect of APIs whose source code is not available. In this paper, we only analyze the APIs in Java 8 standard library, resulting in a list of methods marked as "*may result in side effect*". `ValExtractor` compares method invocations against the list to identify wether the method invocations have side effect. However, such approach may not work for non-Java APIs, local system classes or third-party library classes. In future, we should investigate generic solution for such cases, and the slicing-based approach proposed by Tsantalis and Chatzigeorgiou [45] could be promising. We also notice that the proposed approach depends on type biding provided by Eclipse, and thus if the source code cannot be parsed by Eclipse (e.g., because of incompleteness), it may fail to recognize method invocations.

The proposed approach conducts static analysis to identify potential exceptions caused by extract local variable refactorings. The feature is highly desirable because additional exceptions may result in crash of the software systems. However, the current implementation of the proposed approach covers only *null pointer* exceptions and *class cast* exceptions. In future, we should cover additional exceptions such as *arithmetic* exceptions.

Recommending a name for the newly introduced variable is an indispensable part of extract local variable refactoring. However, the proposed approach simply reuses the implementation of Eclipse JDT. In future, we would like to improve the accuracy in name recommendation for extract local variable refactorings.

7 RELATED WORK

7.1 Improving Reliability of Refactorings

Software refactoring, as well as other changes on complex software applications, is often error-prone. Consequently, researchers have proposed approaches to reduce, avoid, or detect errors introduced by refactorings. For example, to improve the reliability of generalization-related refactorings, Tip et al. [41, 42] leveraged type constraints to verify the preconditions of refactorings, and to determine safe editions of source code. Schäfer et al. [36] and Jonge et al. [4] focused on the reliability of the most widely used refactoring, renaming software entities. The key of their approaches is to create symbolic names that are bound to the renamed software entities by inverting the lookup function of compilers (or by reusing the name analysis provided by compilers), which helps avoid updating (by mistake) references to untouched software entities. Researchers have also provided static analysis-based approaches to improve the reliability of other types of refactorings, including memorization refactorings [51], merging of duplicated code [47], asynchrony refactorings [13, 22], and refactoring for parallelization [19].

As a generic mechanism to improve the reliability of refactorings, preconditions have been widely used to prevent unsafe refactorings [33]. For example, Opdyke [29], one of the best-known pioneers in software refactoring, specified preconditions for various types of refactorings. Tsantalis et al. [44] specified preconditions for move method refactorings. Ubayashi et al. [48] proposed refactoring by contract to verify refactorings based on contracts that are composed of preconditions, postconditions, and invariants described in Contract Writing Language (COW) [38]. Overbey et al. [30] propose differential precondition checking to verify preconditions of refactorings. Refactoring Browser [33] is one of the premiere refactoring tools that support preconditions of refactorings. JRRT [35] is a refactoring engine for JastAddJ. It is well-known for its unique safety checking for software refactorings.

7.2 Detecting Faulty Refactorings

Detecting faulty refactorings is an important and practical way to minimize the risk of software refactorings. Soares et al. [40] proposed `SafeRefactor` that checks for compilation errors in the refactored version first and then generates unit tests for both versions. If any unit test results in conflicting results before and after the refactorings, errors are reported. `SafeRefactorImpact`, proposed by Mongiovi [24], improves SafeRefactor by focusing on only such methods that are impacted by the refactorings. Soares et al. [39] and Melina Mongiovi et al. [23] proposed techniques to identify overly weak (semantic errors) and overly strong conditions (imperfect transformations) in refactoring implementations using SafeRefactor. They identified a number of faulty or imperfect refactorings in Eclipse, NetBeans and other refactoring engines. Gligoric et al. [14] proposed an approach to testing refactoring engines. Using real programs as input, they identified a number of compilation errors caused by refactoring engines of IDEs.

To identify faulty extract variable refactorings, Eilertsen et al. [9] proposed an assertion-based approach. It inserts assertions into refactored programs to validate whether the replaced expressions are equivalent to the variables (that are used to replace the expressions) at the location where the expressions are. Our approach is complementary to theirs in that we prevent unsafe extract variable refactorings whereas they detect faulty refactorings.

8 CONCLUSIONS AND FUTURE WORK

Extract local variable is one of the most popular refactorings. However, manual refactoring is both time-consuming and error-prone. To this end, in this paper, we propose a novel approach ValExtractor to conduct extract local variable refactorings automatically and safely. Our evaluation results on open-source applications suggest that automated extract variable refactorings could be risky, and the state-of-the-practice refactoring tools (both Eclipse and IntelliJ IDEA) did result in a large number of faulty extract variable refactorings that introduced semantic errors. In contrast, ValExtractor successfully avoided all such errors. Our bug reports to Eclipse community have been confirmed and the pull requests implementing our approach have been approved and merged into Eclipse.

It is potentially fruitful to adapt the proposed approach to other programming languages besides Java. It is also practical to parallelize ValExtractor, especially the search algorithm in Algorithm 1 for further speedup.

9 DATA AVAILABILITY

The replication package is publicly available [31].

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers from ESEC/FSE'2023 for their insightful comments and constructive suggestions. This work was partially supported by the National Natural Science Foundation of China (62232003 and 62172037).

REFERENCES

[1] Robert S Arnold. 1986. An introduction to software restructuring. *Tutorial on Software Restructuring* (1986), 1–11.

[2] Block (Eclipse JDT API Specification). 2023. https://help.eclipse.org/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/Block.html.

[3] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. 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 (ESEC/FSE)*. 185–194. https://doi.org/10.1145/1287624.1287651

[4] Maartje De Jonge and Eelco Visser. 2012. A language generic solution for name binding preservation in refactorings. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA)*. 1–8. https://doi.org/10.1145/2427048.2427050

[5] Defects4J – version 1.1.0. 2023. https://github.com/rjust/defects4j/tree/v1.1.0.

[6] B. Du Bois, S. Demeyer, and J. Verelst. 2004. Refactoring - improving coupling and cohesion of existing code. In *11th Working Conference on Reverse Engineering (WCRE)*. 144–151. https://doi.org/10.1109/WCRE.2004.33

[7] Eclipse. 2023. http://www.eclipse.org/.

[8] Eclipse Java development tools (JDT). 2023. https://www.eclipse.org/jdt/.

[9] Anna Maria Eilertsen, Anya Helene Bagge, and Volker Stolz. 2016. Safer refactorings. In *International Symposium on Leveraging Applications of Formal Methods (ISoLA)*. Springer, 517–531. https://doi.org/978-3-319-47166-2_36

[10] Extract/Introduce variable. 2023. https://www.jetbrains.com/help/idea/extract-variable.html.

[11] Martin Fowler. 2018. *Refactoring*. Addison Wesley.

[12] Full List of Preconditions. 2023. https://github.com/liuhuigmail/ValExtractor/#Preconditions.

[13] Keheliya Gallaba, Quinn Hanam, Ali Mesbah, and Ivan Beschastnikh. 2017. Refactoring Asynchrony in JavaScript. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 353–363. https://doi.org/10.1109/ICSME.2017.83

[14] Milos Gligoric, Farnaz Behrang, Yilong Li, Jeffrey Overbey, Munawar Hafiz, and Darko Marinov. 2013. Systematic testing of refactoring engines on real software projects. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7920)*. Springer, 629–653. https://doi.org/10.1007/978-3-642-39038-8_26

[15] Yaroslav Golubev, Zarina Kurbatova, Eman Abdullah AlOmar, Timofey Bryksin, and Mohamed Wiem Mkaouer. 2021. One Thousand and One Stories: A Large-Scale Survey of Software Refactoring. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1303–1313. https://doi.org/10.1145/3468264.3473924

[16] IntelliJ IDEA. 2023. http://www.jetbrains.com/idea/.

[17] ITypeBinding (Eclipse JDT API Specification). 2023. https://help.eclipse.org/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ITypeBinding.html.

[18] Yanjie Jiang, Hui Liu, Xiaoqing Luo, Zhihao Zhu, Xiaye Chi, Nan Niu, Yuxia Zhang, Yamin Hu, Pan Bian, and Lu Zhang. 2023. BugBuilder: An Automated Approach to Building Bug Repository. *IEEE Transactions on Software Engineering* (2023), 1–1. https://doi.org/10.1109/TSE.2022.3177713

[19] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. 2019. Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 619–630. https://doi.org/10.1109/ICSE.2019.00072

[20] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2014. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Transactions on Software Engineering* 40, 7 (2014), 633–649. https://doi.org/10.1109/TSE.2014.2318734

[21] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. 2018. An empirical study on the impact of refactoring activities on evolving client-used APIs. *Information and Software Technology* 93 (2018), 186–199. https://doi.org/10.1016/j.infsof.2017.09.007

[22] Yu Lin, Semih Okur, and Danny Dig. 2015. Study and refactoring of Android asynchronous programming (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 224–235. https://doi.org/10.1109/ASE.2015.50

[23] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Márcio Ribeiro, Paulo Borba, and Leopoldo Teixeira. 2017. Detecting overly strong preconditions in refactoring engines. *IEEE Transactions on Software Engineering* 44, 5 (2017), 429–452. https://doi.org/10.1109/TSE.2017.2693982

[24] Melina Mongiovi, Rohit Gheyi, Gustavo Soares, Leopoldo Teixeira, and Paulo Borba. 2014. Making refactoring safer through impact analysis. *Science of Computer Programming* 93 (2014), 39–64. https://doi.org/10.1016/j.scico.2013.11.001

[25] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2011. How we refactor, and how we know it. *IEEE Transactions on Software Engineering* 38, 1 (2011), 5–18. https://doi.org/10.1109/TSE.2011.41

[26] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. 2013. A comparative study of manual and automated refactorings. In *ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27*. Springer, 552–576. https://doi.org/10.1007/978-3-642-39038-8_23

[27] NetBeans. 2023. http://netbeans.org/.

[28] Online Safe Effect Example and Explanation. 2023. https://github.com/liuhuigmail/ValExtractor/blob/main/Example.md.

[29] William F Opdyke. 1992. *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign.

[30] Jeffrey L Overbey and Ralph E Johnson. 2011. Differential precondition checking: A lightweight, reusable analysis for refactoring tools. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 303–312. https://doi.org/10.1109/ASE.2011.6100067

[31] Replication Package. 2023. https://doi.org/10.5281/zenodo.8276904

[32] ReSharper. 2023. https://www.jetbrains.com/resharper/.

[33] Don Roberts, John Brant, and Ralph Johnson. 1997. A refactoring tool for Smalltalk. *Theory and Practice of Object systems* 3, 4 (1997), 253–263. https://doi.org/10.1002/(SICI)1096-9942(1997)3:4<253::AID-TAPO3>3.0.CO;2-T

[34] Sample Size Calculator. 2023. https://www.surveysystem.com/sscalc.htm/.

[35] Max Schäfer and Oege de Moor. 2010. Specifying and implementing refactorings. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. ACM, 286–301. https://doi.org/10.1145/1869459.1869485

[36] Max Schäfer, Torbjörn Ekman, and Oege De Moor. 2008. Sound and extensible renaming for Java. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications (OOPSLA)*. 277–294. https://doi.org/10.1145/1449764.1449787

[37] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*. 1–11. https://doi.org/10.1145/2393596.2393670

[38] Suguru Shinotsuka, Naoyasu Ubayashi, Hideaki Shinomi, and Tetsuo Tamai. 2006. An extensible contract verifier for AspectJ. In *Proceedings of the 2nd Asian Workshop on Aspect-Oriented Software Development (AOAsia 2)(Workshop at ASE 2006)*. 35–40.

[39] Gustavo Soares, Rohit Gheyi, and Tiago Massoni. 2012. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering* 39, 2 (2012), 147–162. https://doi.org/10.1109/TSE.2012.19

[40] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. 2010. Making Program Refactoring Safer. *IEEE Software* 27, 4 (2010), 52–57. https://doi.org/10.1109/MS.2010.63

[41] Frank Tip, Robert M Fuhrer, Adam Kieżun, Michael D Ernst, Ittai Balaban, and Bjorn De Sutter. 2011. Refactoring using type constraints. *ACM Transactions on Programming Languages and Systems* 33, 3 (2011), 1–47. https://doi.org/10.1007/978-3-540-74061-2_1

[42] Frank Tip, Adam Kiezun, and Dirk Bäumer. 2003. Refactoring for generalization using type constraints. *ACM SIGPLAN Notices* 38, 11 (2003), 13–26. https://doi.org/10.1145/949305.949308

[43] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and removal of type-checking bad smells. In *2008 12th European conference on software maintenance and reengineering (CSMR)*. IEEE, 329–331. https://doi.org/10.1109/CSMR.2008.4493342

[44] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367. https://doi.org/10.1109/TSE.2009.1

[45] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2011. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software* 84, 10 (2011), 1757–1782. https://doi.org/10.1016/j.jss.2011.05.016

[46] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 483–494. https://doi.org/10.1145/3180155.3180206

[47] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. 2015. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1055–1090. https://doi.org/10.1109/TSE.2015.2448531

[48] Naoyasu Ubayashi, Jinji Piao, Suguru Shinotsuka, and Tetsuo Tamai. 2008. Contract-based verification for aspect-oriented refactoring. In *2008 1st International Conference on Software Testing, Verification, and Validation (ICST)*. IEEE, 180–189. https://doi.org/10.1109/ICST.2008.36

[49] Visual Studio. 2023. https://visualstudio.microsoft.com/.

[50] Michael Wahler, Uwe Drofenik, and Will Snipes. 2016. Improving Code Maintainability: A Case Study on the Impact of Refactoring. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 493–501.

https://doi.org/10.1109/ICSME.2016.52

[51] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. 2015. Towards purity-guided refactoring in Java. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 521–525. https://doi.org/10.1109/ICSM.2015.7332506

Received 2023-03-02; accepted 2023-07-27