ORIGINAL ARTICLE

Journal Section

ReInstancer: An Automatic Refactoring Approach for *Instancerof* Pattern Matching

Yang Zhang[1,2*] | **Shuai Hong**[1]

[1]School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang, Hebei 050018, China

[2]Hebei Technology Innovation Center of Intelligent IoT, Shijiazhuang, Hebei 050018, China

Correspondence
Yang Zhang, School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang, Hebei 050018, China
Email: zhangyang@hebust.edu.cn

Present address
[*]School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang, Hebei 050018, China

Funding information
This work was supported in part by the Scientific Research Foundation of Hebei Educational Department under Grant ZD2019093.

The *instanceof* pattern matching is periodically previewed in the latest published JDK by removing the redundant casting and optimizing its usage in different scenarios, which improves the code quality and readability. However, the existing integrated development environment (IDE) does not provide sufficient support for refactoring of *instanceof* pattern matching in all situations. This paper first identifies several cases that cannot be handled by existing IDEs, and then proposes a novel approach called *ReInstancer* to refactor *instanceof* pattern matching automatically. *ReInstancer* conducts visitor pattern analysis for *if* statement blocks with *instanceof*. For the *instanceof* expression, a pattern variable is obtained after removing the redundant type casting by pattern analysis. For multi-branch statements, *ReInstancer* employs class hierarchy analysis and control flow analysis to infer the relationship of instance type in each branch and then optimize these branches. Finally, the multi-branch statements are converted into switch statements or switch expressions. *ReInstancer* is evaluated with 20 real-world applications. The experimental results demonstrate a total of 3558 *instanceof* expressions and 228 multi-branch statements applying the refactoring, and each application takes an average of 10.8s. *ReInstancer* not only reduces the re-

Abbreviations: ReInstancer: An Automatic Refactoring Approach for *Instancerof* Pattern Matching

dundant type casting but also improves the code quality.

KEYWORDS
Automatic refactoring, Pattern matching, instanceof, Program analysis

1 | INTRODUCTION

Pattern matching is the act of checking a given sequence of taken for the presence of constituents of some pattern. It is generally accepted in a variety of programming languages (such as Haskell and ML), text-oriented languages (such as SNOBOL4 and AWK), and object-oriented languages(such as Scala, Java, and C#). Pattern matching uses include outputting the locations of a pattern within a token sequence, outputting some component of the matched pattern, and substituting the matching pattern with some other character sequence. It could provide a convenient way to deconstruct data structures and could enhance language consistency.

Gavin et al.[1] explored the possibility of supporting pattern matching for *instanceof*. The *instanceof* is an operator provided by Java programming languages to determine the instance type of an object variable, and it is normally used pair with type casting. However, the multi-branch statement with *instanceof* requires extensive use of type casting to access object variables or methods, significantly affecting the program's readability and leading to errors. With the release of JDK 14, the first preview of *instanceof* pattern matching was proposed, which avoids redundant type castings by extracting pattern variables. Extracting pattern variables in real-world applications can remove many type castings. This feature is previewed again in JDK 15 and has become a standard feature since JDK16. Since JDK17, another preview feature of converting multi-branch statements together with pattern matching into switch statements or switch expressions is introduced to make the code of pattern matching clean and reliable.

Both academia and industry proposed various techniques for pattern matching and refactoring in the past few years. In the academic community, Wang et al.[2] proposed an approach to program refactoring based on a right-invertible language *Rinv*. They approach refactoring pattern matching smoothly and incrementally. Liu et al. [3] proposed the JMatch language. The JMatch language extends Java with iterable abstract pattern matching, making iteration abstractions convenient. Cantatore [4] designed an algorithm to compare a pattern matching string containing wildcards. However, existing technique is a preliminary exploration of pattern matching to optimize programming languages, and there lacks of prototype implementation. Zhang et al.[5] [6][7][8] [9] [10] presented several concurrency-related refactoring tools to improve lock patterns and detect code smell patterns. In the software industry, refactoring plugins for pattern matching are integrated into the modern integrated development environment (IDE) like Visual Studio and IntelliJ IDEA.

Although many works have been done on pattern matching, we are not aware of any existing research related to *instanceof* pattern matching. The *instanceof* expression is frequently used in large software applications. According to our statistics, more than 1100 *instanceof* expressions exist in the FOP application after searching more than 200 KSLOC. If such a large amount *instanceof* is refactored manually, it not only requires programmers to review the pattern type and extract the pattern variable but also needs a well understanding of the relationship for multi-branch statements. Existing IDEs enable developers to perform pattern matching refactoring by hand. The disadvantage of these IDEs is that the user has to manually select the code that conforms to the pattern matching refactoring by clicking on a pop-up window. When a developer is unfamiliar with some pattern, they would have to look up the documentation, release notes, or Q&A forums to understand how to select the code that conforms to the pattern

matching refactoring. Furthermore, current IDEs do not provide sufficient refactoring support for pattern matching in multi-branch statements, especially in checking the relationships between multiple patterns and refactoring for switch expressions.

In this paper, we introduce an approach, named *ReInstancer*, that could remove redundant type castings by pattern variables and convert multi-branch statements with *instanceof* into switch statements or expressions. Firstly, *ReInstancer* gets the blocks of statements with *instanceof* and multi-branch statements by visitor pattern and performs pattern matching analysis to extract pattern variables. Secondly, *ReInstancer* utilizes program analysis techniques such as class hierarchy analysis and control flow analysis to perform pattern analysis. Finally, the code is refactored according to the analysis results. *ReInstancer* is evaluated on 20 real-world applications, with a total of 3558 *instanceof* expressions applying to refactor and converting 228 multi-branch statements to switch statements or switch expressions. We find that the average of 2.75% reduces the average cyclomatic complexity of the applications after refactoring, which demonstrates the effectiveness of *ReInstancer* in optimizing *instanceof* expressions and multi-branch statements. We also demonstrate the utility of *ReInstancer* by developing a plugin for Eclipse that provides assistance to developers in performing the refactoring.

In summary, the paper makes the following contributions.

- *ReInstancer* analyzes the redundant type casting in the statement block and extracts the required pattern variables.
- We design an analysis algorithm for getting special patterns of multi-branch statements.
- *ReInstancer* is evaluated on several Java benchmarks and applications, demonstrating the effectiveness of *ReInstancer*.
- We empirically evaluated out *ReInstancer* to demonstrate its applicability, effectiveness, trustworthiness, and utility and make our tool publicly available.

This paper extends and refines a previously presented conference paper[11]. Specifically, we extend our algorithm to analyze special patterns for multi-branch statements. Furthermore, we increased the application used for evaluation to verify the generality of *ReInstancer*. Finally, we made a large number of changes related to the presentation of the paper and included more detailed descriptions of the algorithm, evaluation, and related work.

The rest of this paper is organized as follows. Section 2 presents three motivating examples. Section 3 presents our refactoring framework and pattern analysis algorithm. Section 4 shows the experimental evaluation of the refactoring. The related works of literature are examined in Section 5, and conclusions are drawn in Section 6.

2 | MOTIVATION

This section presents three motivating examples to demonstrate the rationale. These examples illustrate a variety of potential problems that are not well solved by prevalent IDEs in pattern matching refactoring.

Figure 1 presents a method *error()* to illustrate the problem of a multi-branch statement with a branch dominated by a branch before it. This method determines the instance type of *o* using a multi-branch statement that checks the type *CharSequence* followed by the type *String*. When we pass an object of type *StringBuffer*, the first branch will always be executed since the *StringBuffer* class implements the *CharSequnece* interface, which introduces dead code when we try to refactor in existing IDEs (e.g. IntelliJ IDEA and Eclipse), which do not report any error. When the problems like this occur in programs, it may lead to a lack of functionality.

Figure 2 presents method *convertToDefaultType()* of class *BinaryType* selected form HSQLDB[12] benchmark. This

```
1 static void error(Object o) {
2    if(o instanceof CharSequence){
3        CharSequence cs=(CharSequence)o;
4        System.out.println("A sequence of length " + cs.length());
5    }else if(o instanceof StringBuffer)
6        StringBuffer s=(StringBuffer)o;
7        System.out.println("A StringBuffer type is dominated: " + s);
8    }else{
9        System.out.println("Other type");
10   }
11}
```

FIGURE 1 The dominated branch

method first determines whether a is *null*, and then determines the instance type of a and executes the statement in the branch. Figure 2(a) presents the original method, which first determines the object a is null or not (lines2-4). Then a multi-branch statement is used to determine the instance type of object *a*. When *a* is *byte[]*, *Float*, or *BigInteger* type, the statements in its branch are executed (lines 5-10). Otherwise, Object a is returned (lines 11-13). Figure 2 (b) shows an improved source code, which merges a *null* test into a switch expression and inserts each branch after the case tag. However, the pattern variables are connected to the arrow symbols similar to the lambda expression[13], and the branches can be optimized similarly to make the code uniform and clean. When we conduct refactoring in modern IDEs, such as IntelliJ IDEA and Eclipse, it does not seem that such refactoring is supported.

```
1 public Object convertToDefaultType(SessionInterface session, Object a){
2     if (a == null) {
3         return a;
4     }
5     if (a instanceof byte[]) {
6         return new BinaryData((byte[]) a, false);
7     } else if (a instanceof Float) {
8         return new Double(((Float)a).doubleValue(),false);
9     } else if (a instanceof BigInteger) {
10        return new BigDecimal((BigInteger)a,false);
11    } else {
12        return a;
13    }
14    throw Error.error(ErrorCode.X_22501);
15}
```

(a) Original source code

```
1 public Object convertToDefaultType(SessionInterface session, Object a){
2     return switch (a){
3         case null ->a;
4         case byte[] a0 ->new BinaryData(a0,false);
5         case BinaryData a0->
6             new Double((a0).doubleValue(),false);
7         case String a0->
8             new BigDecimal(a0,false);
9         default ->a;
10    };
11    throw Error.error(ErrorCode.X_22501);
12}
```

(b) Refactoring for switch expression

FIGURE 2 Converting multi-branch statement into switch expression and integrating *null* test

Figure 3 presents the method *GroundType()* of class *GroundType* selected from CoRed-block project [14]. Figure 3(a) presents a multi-branch statement to determine the instance type of the object type. When the type is of type color, it is required to utilize the *brighter()* method invoked by the pattern variable *co* to determine whether it is character *r*. The more deep level of judgment is guarded by the pattern variable *co* (lines 2-9) and leads to lower code readability. Figure 3(b) demonstrates an improved result that merges two conditional statements into one by using a switch expression. It is easy to infer the source code in Figure 3(b) is cleaner than that in Figure 3(a). However, current IDEs do not provide sufficient support for such refactoring.

3 | REFACTORING FOR *INSTANCEOF* PATTERN MATCHING

In this section, we first present an overview of refactoring for *ReInstancer*. Then, we show the Refactor Prober, the Pattern Matching Analysis, and the algorithms for different patterns. Finally, the refactoring algorithm is introduced.

```java
static void GroundType(Type type,Color color,Shape shape){
    if(type instanceof color) {
         color co=(Color)type;
         if (co.brighter().equals("r")) {
              System.out.println("Color is red");
         }else {
              System.out.println("Color is not red");
         }
    }else if (type instanceof shape) {
         shape s=(shape)type;
         System.out.println("Type is shape");
    }else {
         System.out.println("Other type");
    }
}
```

(a) Original source code

```java
static void testTriangle(Type type,Color color,Shape shape) {
   System.out.println(
     switch (type){
       case color co&&(co.brighter().equals("r"))
                    ->"Color is red";
       case color co ->"Color is not red";
       case shape s  ->"Type is shape";
       default ->"Other type";}
    );
}
```

(b) Source code after refactoring

FIGURE 3 Converting multi-branch statement into switch expression with guarded pattern variable

3.1 | Overview

This section presents *ReInstancer*, a refactoring approach that helps developers identify, localize, and optimize *instanceof* and multi-branch statements. Figure 4 gives an overview of the approach. The approach consists of four parts. First, *ReInstancer* takes the Java source program as input and parses the program to generate an abstract syntax tree (AST) and traverses the AST by the visitor pattern analysis to locate all *if* statement blocks that take *instanceof* as judgment conditions. Second, it checks the statement blocks for type castings matching *instanceof* by pattern matching analysis and gets the pattern variables. Third, it gets the execution pattern of multi-branch statements with *instanceof* by pattern analysis and eliminates the effect of particular patterns. Finally, based on these results, *ReInstancer* removes redundant type casting in the program and converts the multi-branch statement into switch statements or expressions.

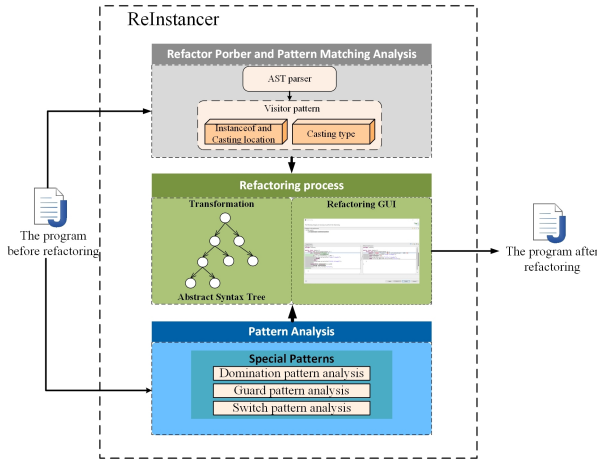

FIGURE 4 Overview of the *ReInstancer* framework

3.2 | Refactor Prober

Refactor Prober is to locate the *if* statement block with *instanceof* as judgment condition and the casting within it. Firstly, *ReInstancer* parses the source program to generate an AST. Secondly, it utilizes the visitor pattern analysis to get all statement blocks with *instanceof* as judgment condition. If the statement block has multiple branches, it is divided into multi-branch statements. Then the visitor pattern is utilized again to get the casting of the statement block. Based on the *instanceof* pattern matching feature, the casting is divided into explicit and implicit casting. Explicit casting is using the direct and specific pattern variable in the source code to request conversion or specify a member from an overloaded class. In implicit casting, the data type is converted automatically.

We will now describe some basic definitions.

Definition 1. *(INSTACNCES JUDGMENT) Given a class object is obj, exists a type T. If the instance type of obj is T, then define obj:=T.*

Definition 2. *(EXPLICIT CASTING AND PATTERN VARIABLE) After the instance judgment, the pattern variable v is defined using T for type casting. T v=(T)obj, calling it an explicit casting and using the ec representation, where v is the pattern variable in the explicit casting. According to pattern matching for instanceof binds instance judgment to a pattern variable v obj:=T v.*

Definition 3. *(THE INSTANCES JUDGMENT) After the instance judgment, type casting utilizing the form of (T)obj is called implicit casting ic.*

Definition 4. *(MULTI-BRANCH STATEMENTS SET) In an program P, given a set MB, collect all multi-branch statements with instanceof, MB={mb_1, mb_2...mb_n }. For $\forall mb_i \in MB$ $(1 \leq i \leq n)$, mb_i={b_{i1}, b_{i2},...b_{ik}}, denotes that mb_i contains k branches, and for $\forall\ b_{ij} \in mb_i$ $(1 \leq i \leq k)$, b_{ij} denotes branch j in mb_i. Since the code in P is finite, MB is a finite set.*

3.3 | Pattern Matching Analysis

Pattern matching analysis is a lightweight way of analyzing whether an instance judgment matches a type casting, like expressions and statement blocks. For Java, the relevant components can be gotten on the AST. If *obj:=T* and an explicit casting are present in the statement block, the pattern variable v is inserted after the type *T*, and the explicit casting is removed to achieve judgment and type casting simply. However, during our collection of programs, we find that when *obj:=T* and the type of *obj* are compatible with *T*, implicit castings are used to access member variables or methods of *obj*. If the statement block has multiple implicit castings, redundant implicit castings can be extracted by adaptive pattern variables.

Specifically, after parsing the AST to get *if* statement block with *instanceof* as judgment condition and the casting within it, the casting can be divided into implicit casting and implicit casting according to the type of the AST node. If the casting is explicit and the object *obj* and type *T* is consistent with the object and type on both sides of *instanceof*, the pattern variable can be obtained at its AST node. When the statement block contains one or more implicit castings consistent with the object *obj* and type *T* of *instanceof*, an adaptive pattern variable can be created in the AST to extract multiple redundant implicit castings.

3.3.1 | Pattern Analysis

Pattern analysis is to get the special patterns of multi-branch statements with *instanceof* by class hierarchy analysis and control flow analysis. The class hierarchy analysis is used to analyze the branches that are dominated in the multi-

branch statement. The control flow analysis is used to analyze the branches, which can be optimized to judge the depth and have a uniform execution pattern in the multi-branch statement.

3.3.2 | Domination pattern analysis

The domination pattern is the dominated branch in the multi-branch statement with *instanceof*. In the multi-branch statement, each branch is used for instance judgment, and when the instance type judged by a branch is the parent class or interface of its subsequent branch, the subsequent branch will be dominated, and the dominated branch will not be executed.

To solve the above problem, the domination pattern analysis utilizes class hierarchy analysis to get the class hierarchy relationship in the source program and build a class hierarchy tree. Then it checks whether the instance type judged by the branch belongs to the class hierarchy tree and finally detects the dependency relationship between instance types of multiple branches. *ReInstancer* utilizes the static analysis tool WALA[15] to get the relationship between classes and classes or between interfaces and classes in the source program and uses the Java reflection mechanism to get the class hierarchy in the Java class library. *ReInstancer* defined the class hierarchy tree.

Definition 5. *(CLASS HIERARCHY TREE) Let CHT be a class hierarchy tree. The tree CHT has one root node. Each node* $t \in CHT$*, has a parent* $p \in CHT$*(except for the root). Each node* $t \in CHT$*, has a list of children. Since all Java inherits from the Object class, the Object class is the root node. Each node t represents the type of class or interface, and the list of children of t store the type of subclasses or implementation classes of t.*

We present the algorithm for the Domination Pattern Analysis in detail. Algorithm 1 gives a pseudocode description of the analysis. Given a multi-branch statement *mb* to be analyzed and the class hierarchy tree *cht* of the source program, we initialize an empty set T to record the *instanceof* instance type t_j. The branch instructions are obtained by traversing the branch b_{ij} of the multi-branch statement *mb*. For each branch *instanceof* instruction, utilize *getType()* to get the instance type (lines 2-5). Finally, iterate through the subsequent branches of b_{ij} and get the instance type of the *instanceof* instruction that is added to the set *T* (lines 6-9). If t_j is a type in the class hierarchy tree or the Java class library, and a subclass type or implementation type of t_j exists in *T*, the branch that is dominated is located (lines 13-14). If this is not the case, the analysis is aborted.

3.3.3 | Guard pattern analysis

The Guard pattern is the branch in the multi-branch statement with *instanceof* that has a deep judgment guarded by a pattern variable. Guard pattern enables other judgments on objects together with pattern matching, which effectively reduces judgment depth and internal call depth.

The Guard pattern analysis gets the control flow instructions in the branch through the intermediate representation (IR) of WALA[15] and optimizes the branch based on the control flow information in the instructions. We define the precondition *c* to specify the types of instructions that need to be satisfied for the guard pattern analysis. For instruction, we use *cb* to denote the conditional branch instruction, *cc* to denote the check cast instruction, *is* to denote the *instanceof* instruction, *v* to denote the pattern variable. The *getUse()* is used to get the value pointed to by instruction. The *getDef()* is used to get the Def value of the instruction.

Definition 6. *(PRECONDITION c)* $\forall\ b_{ij} \in mb_i\ (1 \le i \le k)$*, the set of branch instructions of* b_{ij} *is S=*$\{i_{j1}, i_{j2}, \ldots i_{jn}\}$*, c={bc, is,...cc}. If* $S \in c$*, t=type(cc), getUse(is)=getDef(cb). c includes three types of instructions bc, is, and cc. The c is a subset of S. The type of the instance judged by is is identical to the type of cc, and is is connected to the cb.*

Algorithm 1: Domination pattern Analysis

input : MultiBranchBlock mb,ClassHierarchyTree cht
output: True or False

```
1  T ⟵ ∅;
2  foreach b_ij in mb_i do
3      foreach Instruction i in b_ij do
4          if i is a instanceof statement then
5              t_j ← GetType(i);
6              foreach b_is in mb do
7                  foreach i' in instructions of b_is do
8                      if i' is instanceof statement then
9                          T ⟵ GetType(i');
10                     end
11                 end
12             end
13             if t_j ∈ cht and subtype of t_j ∈ T then
14                 return true;
15             end
16         end
17     end
18 end
19 return false;
```

Algorithm 2 presents a pseudocode of the Guard pattern analysis. Given a multi-branch statement mb_i to be analyzed, *ReInstancer* first traverses each branch b_{ij} of mb_i and then gets the set of branch instructions S of b_{ij} (line 1). When precondition *c* holds, the instance type is identical to the casting type. If the subsequent instruction *i* is in a conditional branch, the instruction type of *i* is a invoke method or access field. *ReInstancer* uses the variable n_i to record the value invoked or accessed in the instruction *cc* and the variable n_v to record the pattern variable *v* in the instruction *i*. When the value of n_i is equal to the value of n_v (lines 3-7), then the deep judgment within the branch is guarded by *v*.

3.3.4 | Switch pattern analysis

The switch pattern is a multi-branch statement with *instanceof* that has a uniform code pattern in each branch, and the last branch is an else block. For example, each branch execution pattern is a method invocation, a variable assignment, or a return statement to get a return value.

The switch pattern analysis first utilizes AST to get the type of statements within the branch and then utilizes control flow analysis to get the dependency of instructions in the branch to obtain the inter-branch execution pattern. We define the switch pattern precondition *sc* to specify the code structure that matches the switch expression.

Definition 7. *(PRECONDITION sc) For* $\forall\ b_{ij} \in mb_i\ (1 \leq i \leq k)$, $b_{ij} \subseteq ec \parallel ic$, $b_{ik} \equiv$ *else block. The sc defines each branch of the multi-branch statement containing the explicit casting ec or an implicit casting ic, and the last branch is an else statement block to ensure the integrity of the multi-branch statement.*

Algorithm 3 gives a pseudocode description of the switch pattern analysis. Given a multi-branch statement *mb* to be analyzed. *ReInstancer* first initializes two empty sets *bfi* and *bsi* for recording instructions (lines 1-2) and then

Algorithm 2: Guard pattern Analysis

```
   input  : MultiBranchBlock mb
   output : True or False
1  foreach b_ij in mb_i do
2      foreach Instruction i in b_ij do
3          if c and i in conditionBranch and i is method invoke or Access Field then
4              n_i ⟵ getRef(cc);
5              n_v ⟵ getDef(i);
6              if n_i==n_v then
7                  return true;
8              end
9          end
10     end
11 end
12 return false;
```

gets the branch instruction set *S* for b_{ij} (lines 3-4). If the precondition *sc* holds, the last and penultimate instructions of the branch are added to the sets *bfi* and *bsi* (lines 5-7). After that, the types of instructions in *bfi* and *bsi* need to be defined for analyzing the execution patterns.

Definition 8. *(RI PATTERN) Multi-branch statement* $\forall\ b_{ij} \in mb_i\ (1 \leq i \leq k)$, *the set of branch instructions* $S=\{i_{j1}, i_{j2},...i_{jn}\}$, *the set of collection instructions is* $RI=\{i_{1n}, i_{2n},...i_{jn}\}.\forall\ i_{jn}\ (1 \leq i \leq k)$, *the instruction type is return instruction.*

Definition 9. *(GI PATTERN)Multi-branch statement* $\forall\ b_{ij} \in mb_i\ (1 \leq i \leq k)$, *the set of branch instructions* $S=\{i_{j1}, i_{j2},...i_{jn}\}$, *the set of collection instructions is* $GI=\{i_{1n}, i_{2n},...i_{jn}\}.\forall\ i_{jn}\ (1 \leq i \leq k)$, *the instruction type is goto instruction.*

Definition 10. *(II PATTERN)Multi-branch statement* $\forall\ b_{ij} \in mb_i\ (1 \leq i \leq k)$, *the set of branch instructions* $S=\{i_{j1}, i_{j2},...i_{j(n-1)}\ i_{jn}\}$, *the set of collection instructions is* $GI=\{i_{1(n-1)}, i_{2(n-1)},...i_{j(n-1)}\}.\forall\ i_{j(n-1)}\ (1 \leq i \leq k)$, *the instruction type is invoke instruction.*

The *RI*, *GI*, and *II* patterns are important guidelines for verifying that branches have uniform execution patterns between branches. When *bfi* is *GI* and *bsi* is *II*. *ReInstancer* first acquires the member and type dependency in the first element of *bsi* using the variables *m* and *t* (lines 12-14). Secondly, *getMemeberReference()* and *getTypeReference()* are used to get the member dependency and type dependency in *bfi*. If the dependency in *bfi* is identical to *m* and *t*, then a uniform execution pattern exists (lines 15-16). Finally, if *bfi* is *RI*, it can be determined that a uniform execution pattern exists because the type of its method only limits the type of the return value (lines 17-19).

3.4 | Refactoring Algorithm

This section introduces the refactoring algorithm of *ReInstancer*. Algorithm 4 gives a pseudocode description main procedure of the refactoring.

ReInstancer first gives the set *MB* of multi-branch statements with *instanceof* and the set *InSet* of *instanceof* statements based on the result of the Refactor Prober. Redundant castings are removed based on pattern matching analysis (lines 2-3). For explicit castings, *ReInstancer* can take the pattern variable from the AST node of the cast statement and bind it to *T* after it (lines 4-6). A redundant castings statement would be removed. Secondly, *ReInstancer* traverses each multi-branch statement mb_i and gets the pattern analysis result by *patternAnalysis()* (line 9). For implicit castings,

Algorithm 3: Switch pattern Analysis

```
input  : MultiBranchBlock mb
output : True or False
1  bfi ⟵ ∅;
2  bsi ⟵ ∅;
3  foreach b_ij in mb do
4      foreach i in instructions S of b_ij do
5          if sc then
6              bfi ← i_n ;
7              bsi ← i_{n-1} ;
8          end
9          if bfi is GI and bsi is II then
10             m ⟵ getMemberReference(bsi[0]);
11             t ⟵ getTypeReference(bsi[0]);
12             if getMemberReference(bsi) == m and getTypeReference(bsi) == t then
13                 return true;
14             end
15         else if bfi is RI then
16             return true;
17         else
18             return false;
19         end
20     end
21 end
```

ReInstancer can detect all implicit castings associated with pattern types on the AST and extract the implicit castings to create adaptive pattern variables. Finally, if mb_i is the domination pattern, information about the dominated branch is given to avoid dead code (lines 10-11). If mb_i is the Guard pattern, a new switch statement is created on the AST node to insert the pattern variable into the case, connect the logical symbols to the branch in the chain merge sequence[13] and replace mb_i with the new switch statement on the AST node (lines 13-16). If mb_i is a switch pattern, a new switch expression is created to insert the branch into the case and replace mb_i on the AST node (lines 18-20).

4 | EVALUATION

This section first introduces the experimental setup and benchmarks, and then presents the research questions and illustrates the experimental results. For the reproducibility of the evaluation, the prototype *ReInstancer* and all benchmarks are available at https://uzhangyang.github.io/research/reinstancer.html

4.1 | Experimental setup and benchmarks

All experiments are conducted on a workstation with 2.4GHZ Intel Core i5 CPU and 8GB main memory. The machine runs Windows 10 and has JDK 17.0.1, Eclipse 4.1.6 ,and WALA 1.5.6 installed.

To evaluate the usefulness of *ReInstancer*, we run it on 20 projects including the HSQLDB[12], Xalan[16] and FOP[17] benchmarks and seventeen real-world applications: Abdra[18], Batik[19], Cassandra[20], Clerezza[21], Deltaspike[22], Ganttproject[23], Hama[24], JBoss[25], Jcommon[26], Jenkins[27], Jhotdraw[28], Johnzon[29], Jsecurity[30], JGroup[31],

Algorithm 4: Refactoring algorithm

```
input  : MB, InstanceofSet InSet
output : Result of the MB refactoring
foreach ins in InSet do
    if ins includes ec then
        get v and remove ec;
    else if ins include ic then
        extract ic and create v;
    end
end
foreach mb_i in MB do
    Result ← PatternAnalysis(mb_i);
    if Result is a Domination pattern then
        Create information for the dominated branch;
    else if Result is a Guard pattern then
        Create switch ;
        Insert v into the case;
        Merge branches to insert switch;
        Replace mb_i with switch;
    else if Result is a Switch pattern then
        Create switch expression;
        Insert v and branch into the case;
        Replace mb_i with switch expression;
end
```

Rhino[32], Oozie[33] and Xerce[34]. HSQLDB is a relational database management system written in Java. Xalan is the open source software library from the Apache project that can transform XML documents into HTML, text, or other XML document types using the XSLT standard stylesheet. FOP is also part of the Apache project, reads a formatting object tree, and renders the resulting pages to a specific output. JGroup is a group communication tool. Jenkins is an open source automation server. Rhino is a JavaScript engine written in Java and open source. The remaining 13 projects are randomly selected open source applications with high visibility in *Github*.

4.2 | Research questions

All evaluate the effectiveness of *ReInstancer* by answering the following questions.

RQ1 How applicable is *ReInstancer*? To be more specific, how many *instanceof* can be refactored? How many multi-branch statements are converted into switch statements or switch expressions?
RQ2 How effective is *ReInstancer* in removing redundant castings by pattern matching?
RQ3 How useful is *ReInstancer* in improving code quality?
RQ4 How many efforts are saved when using *ReInstancer*?
RQ5 Are these refactoring correct?

We answer RQ1 by counting how many *instanceof* pattern matching and multi-branch statements are refactored by *ReInstancer*. We also report the number of *instanceof* that failed to be refactored. RQ2 is answered by how many

redundant castings are removed. To measure the code quality improved by refactoring, RQ3 is evaluated by reporting the average cyclomatic complexity before and after refactoring. We also report the number of modified SLOC. RQ4 is answered by the time spent refactoring each project. RQ5 is answered by inspecting these changes and reporting the possible inference. These numbers approximately estimate the programmer effort that is saved when refactoring with *ReInstancer*.

4.3 | Results

This section presents the experimental results.

4.3.1 | Result for RQ1

For each original project, Table 1 shows the number of *instanceof* and multi-branch statements as well as redundant castings (columns 2-3). For each refactored project, we demonstrate how many *instanceof* and multi-branch statements. We also report how many *instanceof* are not qualified for refactoring. Column 4 in Table 1 shows the number of *instanceof* that can be refactored, while column 5 shows the number of *instanceof* there are not qualified for refactoring. The number of multi-branch statements converted to switch statements and switch expressions is shown in columns 6 and 7. Columns 8 and 9 show the number of switch expressions in the return form and the number of switch expressions in the invoking form.

A total of 3558 *instanceof* is refactored in 20 projects. For FOP, the original project contains a maximum of 1186 *instanceof* and 564 *instanceof* to be refactored, while the Ganttproject and JSecurity projects have a lower number of *instanceof* pattern matching refactoring of 15 and 40. We manually checked these 4203 *instanceof* that are not qualified for refactoring. We find: (1) *instanceof* is located in the assert statement; (2) *instanceof* is only used to judge instances, and no pattern variables exist that can be extracted. The data show that 228 multi-branch statements were refactored in 20 projects, in which 166 were converted to switch statements and 62 were converted to switch expressions. The number of multi-branch statements refactored in Abdera, Fop, HSQLDB, Jenkins, JGroup, Rhino, and Xalan projects is higher at 17, 37, 34, 16, 13, and 36. The programs Ganttproject, Deltaspike, Hama, Jcommon, and Jsecurity were refactored with a lower number of multi-branch statements. For these 62 switch expressions, the number of switch expressions in the form of return is 41, and the number of switch expressions in the form of invoke is 21. We note that the number of *instanceof* and multi-branch statements are higher in projects of the database, server, and software library types, while they are lower in other projects.

The experimental results show that some *instanceof* cannot extract pattern variables for refactoring due to the limitation of casting. From the perspective of the number of *instanceof* and multi-branch statements refactored, *ReInstancer* has a high level of applicability.

4.3.2 | Result for RQ2

To evaluate the effectiveness of *ReInstancer* in removing redundant castings by pattern matching, column 2 in Table 2 shows the redundant castings in the source project. The number of show explicit castings and implicit castings removed after refactoring is shown in columns 3 and 4.

A total of 3673 redundant castings in the 20 projects. The refactoring removed 1319 explicit castings and 2354 implicit castings. For the Fop project, refactoring removed 601 redundant castings, while the Ganttproject project removed 15 redundant castings by refactoring. The number of castings removed for the remaining projects ranged

TABLE 1 Refactoring results of *ReInstancer*

	Before Refactoring		After Refactoring					
Project	#instanceof	#MBI	#RI	#NRI	#RSS	#RSE	#SER	#SEI
Abdera	322	17	140	182	14	3	0	3
Batik	507	7	210	297	4	3	2	1
Cassandra	889	9	335	554	8	1	1	0
Clerezza	143	6	73	70	4	2	2	0
Deltaspike	169	3	83	86	2	1	1	0
FOP	1186	37	564	622	28	9	7	2
Ganttproject	30	1	15	15	1	0	0	0
Hama	73	2	26	47	1	1	1	0
HSQLDB	479	34	249	230	29	5	2	3
JBoss	205	8	115	90	0	8	3	5
Jcommon	130	3	110	20	3	0	0	0
Jenkins	635	16	279	356	11	5	3	2
Jhotdraw	277	10	139	138	9	1	0	1
Johnzon	118	2	66	52	2	0	0	0
Jsecurity	109	3	69	40	1	2	2	0
JGroup	346	11	129	217	10	1	1	0
Oozie	125	4	37	88	4	0	0	0
Rhino	719	13	339	380	9	4	3	1
Xalan	908	36	385	523	21	15	12	3
Xerce	391	6	195	196	5	1	1	0
Total	7761	228	3558	4203	166	62	41	21

from 27 to 395. We find that the number of implicit castings removed after refactoring is much larger than the number of explicit castings. For this situation, we check the FOP project, which has multiple implicit castings in some *if* statement block with *instanceof* as judgment condition and can remove the implicit castings by extracting a pattern variable. As a result, we believe that *ReInstancer* can effectively remove redundant castings and improve the readability of the code.

4.3.3 | Result for RQ3

To answer RQ3, we obtained the SLOC and the average cyclomatic complexity (ACC) from the source code of each project. SLOCCount [35] is employed to obtain SLOC that is used to estimate the developer effort when refactoring with *ReInstancer*. The value of ACC is generated by SourceMonitor [36], which determines the relative complexity of code modules. We also report the number of modified SLOC and the rate of reduction in ACC. The reduction of SLOC indicates the simplified code by refactoring, while ACC is used to measure the code complexity. The bigger the value of ACC is, the worse the code quality is.

The experimental results are summarized in Table 3, which presents the number of SLOC of (column 2) and ACC (column 5) in the original projects. Columns 3 and 6 show the number of SLOC and ACC after refactoring, while columns 4 and 7 show the number of reduced SLOC and the ratio of reduced ACC to #BRA. The data shows that our *ReInstancer* refactoring produces more succinct code. For SLOC, *ReInstancer* reduced the SLOC in the FOP project by up to 833. The other projects reduced SLOC between 46 and 819. Project Johnzon has the lowest number of SLOC reductions of all projects, possibly because of the low number of refactored *instanceof* and multi-branch statements.

TABLE 2 Redundant casting removed by *ReInstancer*

	Before Refactoring	After Refactoring	
Project	#RC	#REC	#RIC
Abdera	143	52	91
Batik	210	60	150
Cassandra	338	170	168
Clerezza	73	15	58
Deltaspike	86	33	53
FOP	601	237	364
Ganttproject	15	6	9
Hama	27	11	16
HSQLDB	271	56	215
JBoss	118	41	77
Jcommon	111	90	21
Jenkins	283	90	194
Jhotdraw	146	60	86
Johnzon	66	10	56
Jsecurity	69	16	53
JGroup	137	32	105
Oozie	38	11	27
Rhino	345	147	198
Xalan	395	125	270
Xerce	200	57	143
Total	3673	1319	2354

The average number of SLOC reductions for each project is 224. For ACC, the *ReInstancer* reduced ACC in the JBoss project by 8.40%, while the Ganttproject and Johnzon projects reduced ACC by the least, possibly because there were fewer refactored multi-branch statements. We examined the JBoss project and found that many multi-branch statements were refactored to switch expressions, and the ACC of the statement was significantly reduced after the refactoring. The average reduction in ACC per project was 2.75%.

We conclude from these results that *ReInstancer* is effective in improving code quality. When it is applied to real-world projects, the generalization of *ReInstancer* is good.

4.3.4 | Result for RQ4

Measuring the developer effort in terms of a precise evaluation is truly difficult. Ideally, we would have observed developers while they refactor and determine how much time is saved with *ReInstancer*. However, the refactoring time may vary depending on the differences in familiarity with *instanceof* pattern matching for different developers.

To approximate estimates of the amount of work saved by *ReInstancer*. We count the number of *instanceof* and multi-branch statements as well as the number of code changes. These figures represent that a developer would have spent time searching for *instanceof* and multi-branch statements to transform the code manually. In total, all projects have 7761 *instanceof* and 228 multi-branch statements that are spread across 20 projects, with an average of 117KSLOC per project. It is labor-intensive to search in such a large Java project to find a small amount of *instanceof* and multi-branch statements and to refactor them. The "#Time" columns in Table 3 report the refactoring time for *ReInstancer*. The automated *ReInstancer* refactoring takes an average of just 10.81 seconds per project. For Cassandra, the refactoring time is 66.5s because its projects are larger and the SLOC is the largest. For Jsecurity, Jcommon, Deltaspike, Rhino, Jhotdraw, and Jhonzon, the refactoring time varies from 1.8s to 5.8s for smaller projects. The refactoring time of *ReInstancer* is mainly used on static analysis, and the refactoring costs more time for larger projects. In

TABLE 3 Comparison before and after refactoring

	SLOC			ACC			#Time
Project	#BRS	#ARS	#RSLOC	#BRA	#ARA	%RACC/#BRA	(s)
Abdera	60273	60002	271	3.9518	3.7625	4.79%	9.6
Batik	191954	191781	173	3.4752	3.3763	2.78%	10.9
Cassandra	462319	462201	128	1.8794	1.8767	0.14%	66.5
Clerezza	15090	14963	127	3.1680	3.1667	0.04%	4.7
Deltaspike	84499	84420	79	2.7800	2.7212	2.11%	3.3
FOP	208968	208135	833	2.7009	2.5983	3.80%	18.2
Ganttproject	21219	21160	59	3.1075	3.1073	0.00%	2.1
Hama	46102	46041	61	2.6244	2.6239	0.02%	3.8
HSQLDB	175568	175081	487	4.0273	3.9729	1.35%	12.2
JBoss	82938	80607	331	3.8305	3.5088	8.40%	8.1
Jcommon	26614	26500	114	2.4292	2.4233	0.24%	2.2
Jenkins	164620	164534	86	2.4962	2.4478	1.93%	14.6
Jhotdraw	80439	80285	154	3.2522	3.2268	0.78%	4.7
Johnzon	48247	48201	46	2.4671	2.4670	0.00%	3.9
Jsecurity	16883	16808	75	2.2042	2.0879	5.28%	1.8
JGroup	122600	122485	115	2.6262	2.5503	2.89%	8.1
Oozie	192468	192404	64	2.8518	2.7600	3.21%	9.2
Rhino	63533	63236	297	4.2688	4.0816	4.34%	5.8
Xalan	141464	140645	819	5.1123	4.7730	6.63%	16.2
Xerce	139653	139483	170	3.9455	3.9186	0.68%	10.3
AVG	117273	117049	224	3.1582	3.0725	2.75%	10.81

combination with the data in Table 3, we find that developer needs to change 224 lines on average, and the refactoring is spread across files.

By contrast, our tool is fully automatic. Even for the largest project with over 400KSLOC, *ReInstancer* can be applied to the whole project in one minute of refactoring. These results show that *ReInstancer* can save a lot of developer effort.

4.3.5 | Result for RQ5

To answer RQ5, we check whether the code after refactoring by *ReInstancer* still has the same behaviors as the original code by running the test cases and manually checking. Specifically, we run the existing developer testing of all 20 projects, and the percentage of passed test cases is 100%. In addition, we manually check all refactoring results. In the process, we manually check 1) if the refactoring changes the behavior of the original code or not; 2)if the refactoring removes the redundant castings; 3) if the usage of a switch statement or switch expression is correct or not; 4)if a switch statement or switch expression is inserted to the right position.

During the inspection, we find that each *instanceof* and multi-branch statement with *instanceof* are based on the result of the analysis(see Section 3), and almost all of them are accurate. However, we also find that a multi-branch statement is refactored to switch statements with missing *default* branches. For example, method *translate()* of class *Stylesheet* in the Xalan project contains a multi-branch statement to determine whether the instance type of *element* is *Tamplate*, *AttributeSet* or *Output*, but the final else statement block contains three comments, and *ReInstancer* refactored it optimally by omitting the last branch. We should note that this case is related to the code structure of

the multi-branch statement and is not related to the correctness of the program and validity of *ReInstancer*.

We do not find any refactorings that change multi-branch statement semantics. More precisely, all multi-branch statements are inserted into the position where they should be. We find that *ReInstancer* did not miss any refactoring. Furthermore, we also inspect the switch statement and switch expression structures and find that all of them are used correctly. These results show that the *ReInstancer* refactoring is correct.

5 | RELATED WORKS

Many researchers have designed refactoring tools to modernize language structures using static analysis techniques. In this section, we investigate refactoring tools for modernizing language structures.

After the JDK1.5 proposal, the standard *java.util.concurrent* library provides a number of data structures and locking constructs. Many difficulties arise when manually transforming a program to use the locking constructs of *java.util.concurrent*, motivating better tool support. The *CONCURRENCER* tool of Dig et al.[37] refactors code to make use of *ConcurrentHashMaps* and *AtomicIntegers*, also provided by the j.u.c library, as well as the refactoring tool[38] *Relooper*, which refactors loops to execute in parallel via the proposed *ParallelArray* class. Wloka et al.[39] presented a mostly-automated refactoring to transform sequential programs to be reentrant, enabling safe parallel execution. Brown et al.[40] proposed *ParaPhrase*, a refactoring tool for generating parallel programs, which increases the programmability of parallel programs. McCloskey et al.[41] presented *Autolocker* to automatically convert the pessimistic atomic section into lock-based code. *Autolocker* retained many of the advantages of optimistic atomic sections and reduced the most burdens of lock-based programming. Schäfer et al.[42] presented a refactoring algorithm to convert the built-in monitor locks into *ReentrantLocks* and *ReentrantRead-WriteLocks*. JDK1.8 introduce several functional features, such as lambda expressions, functional operations, and *Stampedlock*. Gyori et al. [13] presented *Lambdaficator*, which refactors existing Java code to use lambda expressions and function operations to enable parallelism. Zhang et al.[5] proposed an automated refactoring tool *CLOCK* to convert a synchronized lock into a *StampedLock*. *CLOCK* could not only transform read and write locks but also refactor downgrading/upgrading and optimistic read locks. These studies have implemented refactoring tools and automated the tools to modernize the programming language.

The Java language starts to support pattern matching after the release of JDK14. Zhang et al . [43] presented a prototype called *ReSwitcher* that removes the fall-through semantics from switch statements and converts them into switch expressions. *ReSwitcher* provided a preliminary demonstration of the effectiveness of refactoring switch expressions. However, that work merely aims to solve the problem of fall-through semantics. Inspired by this work, our work presented an algorithm to convert multi-branch statements with *instanceof* into switch statements or switch expressions. Some commercial refactoring tools for pattern matching are integrated into modern IDE, like Visual Studio and IntelliJ IDEA. These tools can either recommend refactoring opportunities for pattern matching or extract pattern variables automatically. *ReInstancer* enables Java programmers to benefit from all the advantages reported in these studies.

6 | CONCLUSION

This paper proposes a novel approach called *ReInstancer* to improve code simplicity and quality by pattern matching. We first illustrate several motivating examples that improve the design and then present the analysis and algorithms of refactoring that enable Java developers to simplify *instanceof* expression and multi-branch statements with *instanceof* pattern matching. *ReInstancer* is implemented as the Eclipse plugin and evaluated with 20 real-world applications.

The evaluation shows that a total of 3558 *instanceof* expressions and 228 multi-branch statements applied the refactoring, and each application takes an average of 10.8s. Experimental results provide confidence that the proposed algorithms and implementation can help the developer with refactoring and developer effort. In the future, we will extend this work by improving our approach to match more patterns and using finite automata to optimize multi-branch statements.

Declarations

The authors declare that they have no conflict of interest.

Acknowledgements

The authors would like to thank the insightful comments and suggestions of those anonymous reviewers, which have improved the presentation.

references

[1] Bierman G, JEP 420: Pattern Matching for switch (Second Preview); 2021. https://openjdk.java.net/jeps/420.

[2] Wang M, Gibbons J, Matsuda K, Hu Z. Refactoring pattern matching. Science of Computer Programming 2013;78(11):2216–2242.

[3] Liu J, Myers AC. JMatch: Iterable abstract pattern matching for Java. In: International Symposium on Practical Aspects of Declarative Languages Springer; 2003. p. 110–127.

[4] Cantatore A, Wildcard matching algorithms; 2003. https://xoomer.virgilio.it/acantato/dev/wildcard/wildmatch.html.

[5] Zhang Y, Dong S, Zhang X, Liu H, Zhang D. Automated refactoring for stampedlock. IEEE Access 2019;7:104900–104911.

[6] Zhang Y, Shao S, Zhai J, Ma S. FineLock: automatically refactoring coarse-grained locks into fine-grained locks. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis; 2020. p. 565–568.

[7] Zhang Y. Refactoring-based learning for fine-grained lock in concurrent programming course. Computer Applications in Engineering Education 2022;30(2):505–516.

[8] Zhang Y, Dong C. MARS: Detecting brain class/method code smell based on metric–attention mechanism and residual network. Journal of Software: Evolution and Process 2021;p. e2403.

[9] Zhang Y, Yan J, Qiao L, Gao H. A novel approach of data race detection based on CNN-BiLSTM hybrid neural network. Neural Computing and Applications 2022;p. 1–15.

[10] Zhang Y, Ge C, Hong S, Tian R, Dong C, Liu J. DeleSmell: Code smell detection based on deep learning and latent semantic analysis. Knowledge-Based Systems 2022;255:109737.

[11] Hong S, Zhang Y, Li C, Bai Y. ReInstancer: automatically refactoring for instanceof pattern matching. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings; 2022. p. 183–187.

[12] Group HS, HSQLDB - 100% Java Database; 2021. http://hsqldb.org/.

[13] Gyori A, Franklin L, Dig D, Lahoda J. Crossing the gap from imperative to functional programming through refactoring. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering; 2013. p. 543–553.

[14] CoRed-Block, CoRed-Block; 2022. https://github.com/MurraryZhao/CoRed-Block.

[15] IBM, The t. j. watson libraries for analysis; 2021. http://wala.sourceforge.net/wiki/index.php/Main_Pag.

[16] Apache, Xalan; 2014. http://xalan.apache.org/xalan-j/.

[17] Apache, Fop; 2021. https://xmlgraphics.apache.org/fop/.

[18] Apache, Apache Abdera; 2012. http://abdera.apache.org/.

[19] Apache, The Apache™ Batik Project; 2022. https://xmlgraphics.apache.org/batik/.

[20] Apache, Cassandra; 2021. https://cassandra.apache.org/.

[21] Apache, Welcome to Apache Clerezza; 2022. https://clerezza.apache.org/.

[22] Apache, Apache DeltaSpike; 2020. https://deltaspike.apache.org/.

[23] Thomas A, Ganttproject; 2021. https://www.ganttproject.biz/.

[24] Apache, Apache Hama; 2016. https://hama.apache.org/.

[25] Apache, JBoss-Javassist; 2019. http://www.javassist.org/.

[26] Gilbert D, THE JCOMMON CLASS LIBRARY; 2017. https://github.com/jfree/jcommon.

[27] Kawaguchi K, Jenkins; 2019. https://jenkins.io/.

[28] LGPL, JHotDraw; 2019. https://sourceforge.net/projects/jhotdraw/.

[29] Apache, Apache Johnzon; 2022. https://johnzon.apache.org/.

[30] Apache, Jsecurity; 2009. https://github.com/apache/jsecurity.

[31] Ban B, JGroups; 2021. http://www.jgroups.org/.

[32] Boyd N, Rhino; 2022. https://github.com/mozilla/rhino.

[33] Apache, Apache Oozie Workflow Scheduler for Hadoop; 2021. https://oozie.apache.org/.

[34] Apache, The Apache Xerces™ Project - xerces.apache.org; 2020. http://xerces.apache.org/.

[35] GPL, SLOCCount; 2009. http://www.dwheeler.com/sloccount/.

[36] Software C, SourceMonitor; 2021. https://www.campwoodsw.com/sourcemonitor.html.

[37] Dig D, Marrero J, Ernst MD. Refactoring sequential Java code for concurrency via concurrent libraries. In: 2009 IEEE 31st International Conference on Software Engineering IEEE; 2009. p. 397–407.

[38] Dig D, Tarce M, Radoi C, Minea M, Johnson R. Relooper: refactoring for loop parallelism in Java. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications; 2009. p. 793–794.

[39] Wloka J, Sridharan M, Tip F. Refactoring for reentrancy. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering; 2009. p. 173–182.

[40] Brown C, Hammond K, Danelutto M, Kilpatrick P, Schöner H, Breddin T. Paraphrasing: Generating parallel programs using refactoring. In: International Symposium on Formal Methods for Components and Objects Springer; 2011. p. 237–256.

[41] McCloskey B, Zhou F, Gay D, Brewer E. Autolocker: synchronization inference for atomic sections. In: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages; 2006. p. 346–358.

[42] Schäfer M, Sridharan M, Dolby J, Tip F. Refactoring Java programs for flexible locking. In: Proceedings of the 33rd International Conference on Software Engineering; 2011. p. 71–80.

[43] Zhang Y, Li C, Shao S. ReSwitcher: Automatically Refactoring Java Programs for Switch Expression. In: 2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW) IEEE; 2021. p. 399–400.

Yang Zhang received his Ph.D degree at School of Computer, Beijing Institute of Technology. He was a visiting scholar at Purdue University in 2017. He is currently a professor at School of Information Science and Engineering, Hebei University of Science and Technology. His research interests focus on intelligent software and code refactoring.

Shuai Hong was born is 1998. He is currently a candidate for a Master's degree at Hebei University of Science and Technology. He research interests focus on software refactoring and static analysis.