ReInstancer: Automatically Refactoring for *Instanceof* Pattern Matching

Shuai Hong
School of Information Science and Engineering
Hebei University of Science and Technology
Shijiazhuang, China
hongshuaiss@foxmail.com

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

Chaoshuai Li
School of Information Science and Engineering
Hebei University of Science and Technology
Shijiazhuang, China
li.chaoshuai@foxmail.com

Yu Bai
School of Information Science and Engineering
Hebei University of Science and Technology
Shijiazhuang, China
baiyu@hebust.edu.cn

ABSTRACT

Pattern matching for *instanceof* is widely used with the advantage of conditionally extracting components from objects and with the disadvantage of the compulsory usage of type castings. The feature of pattern matching has been periodically previewed in the latest released version of JDK to avoid redundant type castings and to optimize pattern matching in multi-branch statements. Although pattern matching has many benefits, manual refactoring for *instanceof* pattern matching is time-consuming and tedious. Furthermore, no existing work can provide sufficient support for such refactoring. To this end, this paper presents a refactoring tool *ReInstancer* that mitigates and simplifies type castings by converting a multi-branch statement with *instanceof* pattern matching into a *switch* statement or expression automatically. *ReInstancer* is evaluated on 7 real-world applications, with a total of 4404 *instanceof* expressions. The evaluation results demonstrate that *ReInstancer* can successfully remove 2060 type castings and refactor 141 multi-branch statements with a total of 1972 *instanceof* pattern matching, which improves the code quality.

CCS CONCEPTS

• **Software and its engineering → Software evolution**.

KEYWORDS

Refactoring, Pattern matching, Switch expression, Instanceof, Program analysis

*Corresponding author: Yang Zhang (zhangyang@hebust.edu.cn).

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516868

ACM Reference Format:
Shuai Hong, Yang Zhang, Chaoshuai Li, and Yu Bai. 2022. ReInstancer: Automatically Refactoring for *Instanceof* Pattern Matching. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516868

1 INTRODUCTION

Pattern matching is the act of checking a given sequence of tokens for the presence of the constituents of some pattern. It is generally accepted in a variety of programming languages including functional languages (such as Haskell and ML), text-oriented languages (such as SNOBOL4 and AWK), and object-oriented languages (such as Scala, Java, and C#).

JDK14 proposes the first preview feature on pattern matching for *instanceof* to avoid redundant type castings by extracting pattern variables. Extracting pattern variables in real-world applications can remove a lot of explicit castings and implicit castings. This feature is previewed again in JDK15 and has become a standard feature since JDK16. Pattern matching for *instanceof* owns the advantage of conditionally extracting components from objects and the disadvantage of the compulsory usage of type castings. Since JDK17, another preview feature of converting multi-branch statements together with pattern matching into *switch* statements or expressions is introduced to make the code of pattern matching clean and reliable.

Both academic and industrial researchers proposed various refactoring techniques and tools for pattern matching in the past few years. In the academic community, Wang et al. [23] introduced an approach to refactor pattern matching smoothly and incrementally. AlOmar et al. [1] proposed a technique to decouple patterns from concrete representation to optimize the code quality. Dig et al. presented refactoring tools [9][14][17][18][19] to modernize language constructs in Java and C# programs. Several empirical studies [12][16][20][21] conducted refactoring on pattern matching of the libraries or language constructs. Zhang et al. presented several concurrency-related refactorings to improve lock patterns [25][27] and detect code smell patterns [24]. Our previous work [26] provided a preliminary demonstration of the effectiveness of refactoring switch expressions. However, that work merely aims to

solve the problem of fall-through semantics. Furthermore, it does not consider how to convert pattern matching for *instanceof*. In the industrial setting, refactoring tools for patterning matching are integrated into the modern integrated development environment (IDE) like Visual Studio and IntelliJ IDEA. These tools can either recommend refactoring opportunities for pattern matching or extract pattern variables automatically.

Although many techniques and tools are proposed for pattern matching refactoring, we are not aware of any works that conduct refactoring to convert multi-branch statements with pattern matching into switch statements or expressions. Manual refactoring for pattern matching is tedious and error-prone. According to our statistic, more than 1,100 *instanceof* pattern matching exist in FOP application [6] after searching more than 200 KSLOC. If such a large amount of pattern matching is refactored, it not only requires programmers to review the pattern type and extract pattern variables but also needs a well understanding of the relationship for multi-way statements. 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.

This paper proposes an automated refactoring tool, called *ReInstancer*, to remove redundant type castings by adaptive pattern variables and to convert multi-branch statements with pattern matching into *switch* statements or expressions. *ReInstancer* is developed as an Eclipse plug-in[1], and is evaluated on 7 real-world applications, with a total of 4404 *instanceof* expressions. The experimental results show that *ReInstancer* removes 2060 type castings and converts 141 multi-branch statements into 109 *switch* statements as well as 32 *switch* expressions, with a total of 1972 *instancesof* pattern matching. We find that the average cyclomatic complexity of the applications after refactoring is reduced by an average of 3.83%, which demonstrates the effectiveness of *ReInstancer* in improving code quality.

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()* published in the *OpenJDK* [8] to illustrate the problem of domination pattern. This method matches the type *o* by means of a *switch* statement in which it checks the type *CharSequence* followed by type *String*. When we pass an object of type *String*, the first *case* statement will be always executed since the *String* class implements the *CharSequence* interface, which introduces dead code. This example demonstrates that a pattern type is dominated by another pattern type that precedes to it. Existing IDEs such as IntelliJ IDEA and Eclipse do not report this kind of potential threats to code vulnerability.

Figure 2 presents method *convertToDefaultType()* of class *BinaryType* selected from *HSQLDB* project [13]. Figure 2(a) presents the original method in which it first judges whether or not the parameter *a* is null (Line 2) and then conducts pattern matching (Lines 5-13). Figure 2(b) shows an improved implementation by converting this multi-branch statement into a *switch* expression. It merges a *null* testing into the *switch* expression, thus makes the code unified 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]ReInstancer is available at http://uzhangyang.github.io/research/reinstancer.html

```
1 static void error(Object o) {
2    switch(o) {
3       case CharSequence cs ->
4              System.out.println("A sequence of length " + cs.length());
5       case String s ->
6              System.out.println("A String type is dominated: " + s);
7       default -> {
8              break;
9       }
10   }
11}
```

Figure 1: The domination pattern and dominated pattern label

```
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 BinaryData) {
8           return castOrConvertToType(session, a,Type.SQL_VARCHAR,false);
9     } else if (a instanceof String) {
10         return castOrConvertToType(session, a,Type.SQL_VARCHAR,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 ->
6              castOrConvertToType(session,a,Type.SQL_VARCHAR,false);
7        case String ->
8              castOrConvertToType(session,a,Type.SQL_VARCHAR,false);
9        default ->a;
10    };
11    throw Error.error(ErrorCode.X_22501);
12}
```

(b)Refactoring for switch expression

Figure 2: Converting multi-branch *if* statement into *switch* expression and integrating *null* test

Figure 3 illustrates a method *testTriangle()* selected from *OpenJDK* [8]. Figure 3(a) presents the original source code that includes the multi-branch statement with guard pattern. It leverages multiple *instanceof* expressions to judge the actual type of variable *s*. Different from the example in Figure 2(a), it further checks whether or not the area of *s* is larger than 100 when the type of *s* is *Triangle*. The latter is guarded by the former. 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 a sufficient support for this kind of refactoring.

3 DESIGN

Figure 4 presents an overview of the design of *ReInstancer*. Our approach takes the Java project with *instanceof* pattern matching as input. It first parses the program to generate an abstract syntax tree

```java
static void testTriangle(Shape s) {
    if(s instanceof Triangle) {
        Triangle t=(Triangle)s;
        if (t.calculateArea() > 100) {
            System.out.println("Large triangle");
            break;
        }else {
            System.out.println("Little triangle");
        }
    }else if (s instanceof Circle) {
        Circle c=(Circle)s;
        System.out.println("Circle");
    }else {
        System.out.println("Possibly a small triangle");
    }
}
```

(a)Original source code

```java
static void testTriangle(Shape s) {
   System.out.println(
     switch (s){
       case Triangle t && (t.calculateArea() > 100)
                        ->"Large triangle";
       case Triangle t ->"Little triangle";
       case Circle c ->"Circle";
       default ->"Possibly a small triangle";}
   );
}
```

(b) Source code after refactoring

Figure 3: Converting multi-branch *if* statement into *switch* expression with guard pattern

(AST) and then traverses the AST by the visitor pattern analysis to locate all *if* statements that take pattern matching as judgment condition, as well as the type castings for this pattern. Then a lightweight static analysis based on Wala [15] is applied. Through control flow analysis and class hierarchy analysis, structural information is collected for multi-branch statements. Pattern analysis involves the optimization of judgment conditions to effectively reduce judgment depth and internal call depth. Finally, *ReInstancer* removes redundant casting in the program and refactors the multi-branch *if* statements into *switch* statements or expressions.

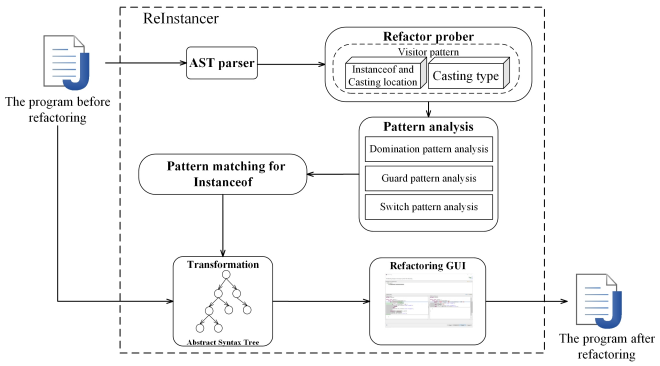

Figure 4: Overview of *ReInstancer*

3.1 Refactor prober

The function of refactor prober is to locate all *instanceof*-based multi-branch statements as well as type casting controlled by *instanceof* statements. Also, it handles both explicit and implicit type casting. To be more specific, we first generate an AST using Eclipse JDT[10] for each Java file, and then leverage visitor patterns to locate them. *ReInstancer* can obtain the object that is on the left of the *instanceof* keyword and the type on the right side when traversing the AST. If the castings within the statement match the object and the type judged by *instanceof*, *ReInstancer* can get the parent node of the castings on the AST and analyze the type casting at its parent AST node to classify it as explicit casting or implicit casting.

3.2 Pattern analysis

ReInstancer locates multi-branch statements with *instanceof* on AST and then leverages Wala [15] to conduct domination pattern analysis, guard pattern analysis, and switch pattern analysis on these statements. The results are used to guide the refactoring of multi-branch statements to *switch* statements or expressions.

Domination pattern analysis. When multiple types are matched simultaneously in the multi-branch statement, domination pattern analysis is carried out to check the correctness of pattern matching. A domination pattern means that a branch is dominated by its sub-branch when inherited relations and implementation relations are presented. As a result, the dominated branch will never have a chance to be executed. Therefore, it is necessary to detect the dependant relationship between multiple branches in a multi-branch statement. Domination pattern analysis is done by class hierarchy analysis and reflection mechanism.

Guard pattern analysis. Guard pattern enables other judgments on objects together with pattern matching, which effectively reduces judgment depth and internal call depth. We perform guard pattern analysis before multi-branch statements are refactored. We analyze the control flow of multi-branch statements to detect the existence of extended judgments corresponding to pattern matching inside the statement, such as rule calculations and relational constraints, etc. After that, we optimize the judgment conditions of multi-branch statements by combining pattern matching with judgments logically. It would be refactored to guard pattern during the refactoring to generate *switch* statements or expressions. For cases where the object needs to be determined as *null* firstly, *ReInstancer* can automatically combine this identification with multi-branch statements as well to improve code simplicity.

Switch pattern analysis. *ReInstancer* analyzes each branch of multi-branch statements in a control flow graph and obtains the code pattern (such as method invocation and variable assignment, etc.) for each branch in the AST. If each branch has consistent code patterns, we convert it into *switch* expression. Otherwise, we convert it to a *switch* statement which is compatible with inconsistent code patterns.

3.3 Pattern matching for Instanceof

ReInstancer provides refactoring for the explicit castings and implicit castings in pattern matching based on the results of the refactor prober. For explicit castings, *ReInstancer* can take the pattern variable directly from the casting statement and apply it to the *instanceof* expression. Redundant type casting statements would be removed at the same time. For implicit castings, *ReInstancer* still needs to combine the pattern type and the result of static analysis to extract pattern variables. By analyzing the control flow of the

Table 1: Refactoring results of *ReInstancer*

	Before refactoring	After refactoring						
Project	#instanceof	#RL	#REC	#RIC	#MBI	#RSS	#RSE	%RR
Cassandra	889	335	170	168	9	8	1	37.7%
FOP	1186	564	237	364	37	28	9	47.6%
HSQLDB	479	249	56	215	34	29	5	52.0%
JGroup	346	129	32	105	11	10	1	37.3%
JBoss	205	115	41	77	8	8	0	56.1%
Xalan	908	385	125	270	36	21	15	42.4%
Xerce	391	195	57	143	6	5	1	49.9%
Total	4404	1972	718	1342	141	109	32	44.8%

pattern matching, *ReInstancer* can detect all implicit castings related to pattern types and replace them with adaptive pattern variables.

For multi-branch statements without any domination pattern, *ReInstancer* first performs pattern matching for each branch to get pattern variables and remove redundant type castings. And then, the judgment logic related to the pattern variables in each branch is integrated with the case label based on the results of guard pattern analysis. Finally, the results of switch pattern analysis are used to determine whether the multi-branch statements are converted to switch expressions or to switch statements.

3.4 Transformation

ReInstancer conducts transformation on AST by converting multi-branch statements into *switch* statements or expressions and removing redundant explicit and implicit type casting. *ReInstancer* obtains the AST nodes of the single-branch statement, multi-branch statement, and type castings on the AST. For single-branch statements, *ReInstancer* inserts the AST node of its pattern variable after the instance type and removes the type casting. For multi-branch statement, *ReInstancer* first creates a new *switch* statement or expression based on the result of pattern analysis, then insert the pattern variable into the case label and removes the type casting, and finally inserts the *switch* statement or expression into the AST node of the multi-branch statement and removes the original multi-branch statement.

4 EVALUATION

We implement a prototype *ReInstancer* leveraging Wala, and the prototype is integrated as a plug-in of Eclipse, and we empirically evaluate it to address the following question:

RQ1: How effective is *ReInstancer* in refactoring *instanceof* pattern matching?

RQ2: How useful is *ReInstancer* in improving code quality?

All experiments are conducted on a workstation with 2.4GHz Intel Core i5 CPU and 8GB main memory. The operating system is Windows 10, and the JDK version is 17.0.1. We evaluate *ReInstancer* on 7 real-world applications including Cassandra[5], FOP[6], HSQLDB[13], JGroup[7], JBoss[3], Xalan[2] and Xerces[4].

4.1 Effectiveness in Refactoring

We evaluate the effectiveness of *ReInstancer* on these projects whose size varies from 82,938 to 462,329 source lines of code (SLOC), which indicates our prototype is generally applicable to large projects.

The evaluation results are tabulated in Table 1, which presents the project (column 1), the numbers of *instanceof* expressions in the original projects (column 2), and the data after refactoring (columns 3-9). Column 3 shows the number of refactored *instanceof* expressions (#RL). Columns 4 and 5 show the number of reduced explicit casting (#REC) and reduced implicit casting (#RIC), respectively. The number of refactored multi-branch statements (#MBI) is presented in Column 6. Columns 7 and 8 present the number of refactored multi-branch statements that converted into *switch* statement (#RSS) and the number of those statements that converted into *switch* expression (#RSE). The last column shows the percentage of successful refactoring between #RL and #instanceof.

ReInstancer refactors a total of 1972 *instanceof* expressions with an average of successfully refactoring rate is 44.8%. We should note that 2432 *instanceof* expressions are not refactored among all projects. We manually examine them and find that these *instanceof* expressions are only used to judge instances without any redundant type casting. 718 explicit castings and 1342 implicit castings are reduced. A total of 141 multi-branch statements is converted to 109 *switch* statements and 32 *switch* expressions.

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 tests of all 7 projects, and then the percentage of passed test cases is 100%. In addition, we manually check all refactoring results. In the process, we manually check the following aspects: 1) if the refactoring changes the behaviors of original code or not; 2) if the usage of a *switch* statement or expression is correct or not; 3) if a *switch* statement or expression is inserted to the right position. The results demonstrate that all refactored multi-branch statements with *instanceof* are correct.

4.2 Improving code quality

To answer RQ2, we obtain the reduction in SLOC by the tool SLOCCount [11] and the average cyclomatic complexity (ACC) by the tool *SourceMonitor* [22]. The reduction of SLOC indicates the simplified code by refactoring, while ACC is used to measure the code

Table 2: Comparison before and after refactoring

	Before refactoring		After refactoring				
Project	$\#SLOC_1$	$\#ACC_1$	$\#SLOC_2$	$\#RSLOC$	$\#ACC_2$	$\#RACC$	$\%\#RACC/\#ACC_1$
Cassandra	462329	1.8794	462201	128	1.8767	0.0027	0.14%
FOP	208968	2.7009	208135	833	2.5983	0.1026	3.80%
HSQLDB	175568	4.0273	175081	487	3.9729	0.0544	1.35%
JGroup	122600	2.6262	122485	115	2.5503	0.0759	2.89%
JBoss	82938	3.8305	82607	331	3.5088	0.3217	8.40%
Xalan	141464	5.1123	140645	819	4.7730	0.3393	6.64%
Xerce	139653	3.9455	139483	170	3.9186	0.0269	0.68%
AVG	190503	3.4460	190091	412	3.3140	0.1319	3.83%

complexity. The bigger the value of ACC is, the worse the code quality is.

The results are summarized in Table 2, which presents the number of SLOC (column 2) and ACC (column 3) in the original projects, and the data after refactoring. Columns 4 to 7 show the number of SLOC, the number of reduced SLOC, the number of ACC, and the number of reduced ACC, respectively. Column 8 demonstrates the ratio between $\#RACC$ and $\#ACC_1$. The number of #RSLOC is ranging from 115 to 833, with an average of 412 #RSLOC. The ACC is reduced by up to 8.40 % and with an average of 3.83%. The results demonstrate that the quality of these projects refactored by *ReInstancer* can be improved.

5 CONCLUSIONS

ReInstancer has been designed to improve code simplicity and quality by pattern matching refactoring. This paper first illustrates several motivating examples that might improve the design, and then present the analysis and refactoring that enable Java developers to simplify multi-branch *if* statements with *instanceof* pattern matching. *ReInstancer* is implemented as an Eclipse plug-in and evaluated with 7 real-world projects. Experimental results show that *ReInstancer* can effectively refactor multi-branch *if* statement with *instanceof* to *switch* statement or expression and remove redundant type casting by pattern matching. Future works include that we will continue to evaluate our tool on more projects and to improve the tool to match more patterns.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This work was supported in part by the National Natural Science Foundation of China under Grant 61902108, in part by the Scientific Research Foundation of Hebei Educational Department under Grant ZD2019093, in part by the Natural Science Foundation of Hebei Province under Grant F2019208305, and in part by Postgraduate Innovative Research Foundation of Hebei Educational Department under Grant CXZZSS2022081.

REFERENCES

[1] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2019. Do design metrics capture developers perception of quality? an empirical study on self-affirmed refactoring activities. *arXiv preprint arXiv:1907.04797* (2019).
[2] Apache. 2014. *Xalan.* http://xalan.apache.org/xalan-j/
[3] Apache. 2019. *JBoss-Javassist.* http://www.javassist.org/
[4] Apache. 2020. *The Apache Xerces™ Project - xerces.apache.org.* http://xerces.apache.org/
[5] Apache. 2021. *Cassandra.* https://cassandra.apache.org/
[6] Apache. 2021. *Fop.* https://xmlgraphics.apache.org/fop/
[7] Bela Ban. 2021. *JGroups.* http://www.jgroups.org/
[8] Gavin Bierman. 2021. *JEP 420: Pattern Matching for switch (Second Preview).* https://openjdk.java.net/jeps/420
[9] Danny Dig. 2010. A refactoring approach to parallelism. *IEEE software* 28, 1 (2010), 17–22.
[10] Eclipse. 2021. *Eclipse Java development tools.* https://www.eclipse.org/jdt/
[11] GPL. 2009. *SLOCCount.* http://www.dwheeler.com/sloccount/
[12] Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. 2010. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement.* 1–10.
[13] Hypersonic SQL Group. 2021. *HSQLDB - 100% Java Database.* http://hsqldb.org/
[14] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. 2013. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* 543–553.
[15] IBM. 2021. *The t. j. watson libraries for analysis.* http://wala.sourceforge.net/wiki/index.php/Main_Pag
[16] Siim Karus and Harald Gall. 2011. A study of language usage evolution in open source software. In *Proceedings of the 8th Working Conference on Mining Software Repositories.* 13–22.
[17] 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.
[18] Yu Lin, Cosmin Radoi, and Danny Dig. 2014. Retrofitting concurrency for android applications through refactoring. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 341–352.
[19] Semih Okur, David L Hartveld, Danny Dig, and Arie van Deursen. 2014. A study and toolkit for asynchronous programming in C#. In *Proceedings of the 36th International Conference on Software Engineering.* 1117–1127.
[20] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2013. Adoption and use of Java generics. *Empirical Software Engineering* 18, 6 (2013), 1047–1089.
[21] R Robbes, E Tanter, and D Rothlisberger. 2011. How developers use the dynamic features of programming languages: the case of smalltalk. In *Proceedings of the International Working Conference on Mining Software Repositories*, Vol. 10.
[22] Campwood Software. 2021. *SourceMonitor.* https://www.campwoodsw.com/sourcemonitor.html
[23] M Wang, J Gibbons, K Matsuda, and Z Hu. 2013. Refactoring Pattern Matching. *Science of Computer Programming* 78, 11 (2013), 2216–2242.
[24] Yang Zhang and Chunhao Dong. 2021. MARS: Detecting Brain Class/Method Code Smell Based on Metric-Attention Mechanism and Residual Network. In *Journal of Software: Evolution and Process.*
[25] Yang Zhang, Shicheng Dong, Xiangyu Zhang, Huan Liu, and Dongwen Zhang. 2019. Automated refactoring for stampedlock. *IEEE Access* 7 (2019), 104900–104911.
[26] Yang Zhang, Chaoshuai Li, and Shuai Shao. 2021. ReSwitcher: Automatically Refactoring Java Programs for Switch Expression. *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2021), 400–401.
[27] Yang Zhang, Shuai Shao, Juan Zhai, and Shiqing Ma. 2020. FineLock: automatically refactoring coarse-grained locks into fine-grained locks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 565–568.