Code Transformation Issues in Move-Instance-Method Refactorings

Jongwook Kim  
Iona College  
jkim@iona.edu

Don Batory  
University of Texas at Austin  
batory@cs.utexas.edu

Milos Gligoric  
University of Texas at Austin  
gligoric@utexas.edu

Abstract—Refactorings, by definition, preserve the behavior of a target program. Such a strong semantic property is encoded by a set of preconditions for each refactoring. Only if all preconditions are satisfied will a target program be transformed. The code transformation that implements the refactoring follows another set of rules to produce syntactically-correct, refactored code. Consequently, it is easy to believe that most behavior-changing violations in refactorings are induced by incorrect preconditions or lack of required checks. In this paper, however, we show that code transformations for Move-Instance-Method Refactoring available in several popular Java Integrated Development Environments do not preserve program behavior. We report these errors and propose solutions for each identified problem.

I. INTRODUCTION

Work on software quality improvement that uses refactorings [1]–[3], like finding refactoring opportunities [4]–[14], assumes refactorings preserve program behavior. Using off-the-shelf Integrated Development Environment (IDE) refactorings implicitly adopts frailties of IDE refactoring engines. Refactorings are believed to be built on rock-solid foundations; to our chagrin, this is not so [15], [16].

Among primitive refactorings, a Move-Instance-Method Refactoring (MiMR) affects many different types of program elements by applying code transformations on modifiers, method signatures, method calls, and method bodies. It is also among the most commonly used and representative refactoring of today’s IDEs [17], [18].

In the last few years, we extensively used the Eclipse Java Development Tools (JDT) refactoring engine [19], [20] and (as a consequence of the problems that we found) have built a new refactoring engine for Java [21]. In doing so, we have become aware of the limitations of IDE refactoring engines, and appreciative of the difficulties to meet the high standards expected of refactorings, namely behavior preservation.

In this paper, we distill our experiences. We investigate how MiMR transforms code in the latest versions of Eclipse JDT [22], IntelliJ IDEA [23], Apache NetBeans [24], and Oracle JDeveloper [25] IDEs. Regardless of preconditions that these Java IDEs check, we found all may change program behavior due to incorrect code transformation rules.

II. MOVE INSTANCE METHOD

A MiMR moves a non-static (instance) method from one class declaration to another. The method can be moved via an existing parameter or via a field variable, whose type (a user-defined class) becomes the destination of the moved method. Figures 1 and 2 show typical examples where method m is moved from class A to class B via a parameter and a field variable, respectively.

Table I lists 20 preconditions for MiMR and shows which are considered by each Java IDEs. The complete set of MiMR (and other refactoring) preconditions is unknown; we manually created the list for JDT whose source code and error messages are publicly available (the latest JDT skips preconditions #6 and #11 in Table I). We also used JDT’s regression test examples to fill the precondition checks of other IDEs in Table I.

This table shows these Java IDEs use different sets of preconditions for MiMR. A consequence is that it is easy to find
examples where program behavior is not preserved [26], [27]. Despite relatively stronger precondition checks in JDT, we and others continue to discover incorrect preconditions [15], [16], [19], [21].

Precondition checks filter refactorings that may change program behavior. If all checks pass, the code is transformed into refactored code without syntactic errors. However, flaws in code transformation rules in Java IDEs can also break behavior preservation in MiMR, which we consider next.

III. CODE TRANSFORMATION FLAWS

A. Parameter Optimization

When an instance method is moved, an extra parameter is added to reference members of the origin class. All four IDEs apply parameter optimization [19] in MiMR, meaning that the extra parameter is not added when the target method’s body does not reference members of the origin class.

![Diagram](a) Before - Parameter Optimization in Move A.m(B) to B.m()

![Diagram](b) After moving m

Figure 3 illustrates parameter optimization. A consequence is that the class creation expression new A() in Figure 3a is removed. If there are side-effects in the prefix expression of a target method call (such as new A()), its elimination alters program behavior [28]. To be fair, checking for side-effects is a complicated and expensive analysis [30]–[35].

B. Swapping Prefix and Parameter Expressions

When a parameter (not a field variable) type is used as the destination in a MiMR, the prefix expression of each method invocation becomes the value of a newly-introduced origin-type parameter. Also, the value of the parameter used as destination in a method invocation becomes the prefix expression after move. Figure 4 shows two class creation expressions new A() and new B() are swapped after a move.

![Diagram](a) Before - Swapping Prefix & Parameter Expressions in Move A.m(B,C,D) to B.m(A,C,D)

![Diagram](b) After moving m

The same problem occurs when Change-Method-Signature Refactoring (CMSR) removes an existing parameter, thereby removing the corresponding expression in a method call [29]. CMSR in JDT, NetBeans and JDeveloper fail to check this precondition.
In JDT’s MtMR, the new origin-type parameter takes the position where the destination-type parameter existed before the move. This means the position of a new parameter is determined by the position of the destination-type parameter as in Figure 4. The other three IDEs work differently by placing the new parameter last; doing so alters the signature of method \( m \) in Figure 4b to:

\[
\begin{align*}
(1) & \quad \text{void } m(C c, D d, A a) \{ \ a.i = 0; \ }
\end{align*}
\]

When a method call expression is executed in Java, the prefix of a method call is evaluated first and then parameter expressions are evaluated in left-to-right order. Therefore, swapping the prefix and parameter expressions changes their evaluation order, which may change program behavior if there are side-effects [36].

NetBeans has a serious bug that always uses this keyword to reference the origin-class members regardless of a prefix expression. So method \( n \) in Figure 4a is transformed to (2) after the move. The last argument should have the value of new \( A() \).

\[
\begin{align*}
(2) & \quad \text{void } n() \{ \\
& \quad \text{new } B().m(new\ C(), new\ D(), \text{this}); \}
\end{align*}
\]

C. Dereferenced Null

When a method call explicitly lists null as the destination argument, the method cannot be moved as null becomes the prefix expression after move. NetBeans does not check this precondition #13 in Table I.

\[
\begin{align*}
\text{class } A\{ \\
& \quad \text{int } i = 0; \\
& \quad \text{void } m(B b) \{ \\
& \quad \quad \text{i++;} \\
& \quad \}
& \quad \text{void } n() \{ \\
& \quad \text{B b = null;} \\
& \quad \text{m(b);} \\
& \quad \}
& \quad \text{class } B\{
\end{align*}
\]

(a) Before

Fig. 5. Dereferenced Null in Move \( A.m(B) \) to \( B.m(A) \)

A more challenging problem is to know whether the destination parameter’s value (which becomes the prefix expression after a move) evaluates to null [38] as in Figure 5. There are analyses to determine dereferenced nulls [39]–[44] but we are unaware if null-analyses have been used in refactoring engines. We do know that only JDT warns about the possible (not all) conditions of null pointer access as in Figure 5b.

D. Duplicated Prefix Expression

When a method is moved via a field variable, a different code transformation rule is applied (recall Figure 2). In a reference to the moved method, the prefix expression is not exchanged with the destination parameter’s expression (Section III-B). Instead, a field variable name used as the destination is added to the prefix. Also, a new origin-type parameter is introduced as described in Section III-A. The new parameter’s value is a copy of the prefix expression in a method call. This means that the prefix expression is evaluated twice at run-time after a move [45] (see Figure 6). Even when a field variable is used as a destination, NetBeans still transforms method invocations incorrectly as in (3). It drops origin prefix new \( A() \) and uses this keyword to reference the origin-class members.

\[
\begin{align*}
(3) & \quad \text{void } n() \{ \ b.m(this); \}
\end{align*}
\]

\[
\begin{align*}
\text{class } A\{ \\
& \quad \text{int } i = 0; \\
& \quad \text{void } m(C c, D d, A a) \{ \ a.i = 0; \}
& \quad \text{class } B\{
\end{align*}
\]

(a) Before

(b) After moving \( m \)

Fig. 6. Duplicated Prefix Expression in Move \( A.m() \) to \( B.m(A) \)

IV. PROPOSED SOLUTIONS

We propose solutions to each of the flaws discussed earlier. We believe our solutions are (or can be made) practical.

**Parameter Optimization** is a failure of separation of concerns. Parameter optimization is a distinct refactoring that is bundled with MtMR and should be unbundled. It should be applied only when the prefixes of all method invocations are simple variables, which are free from side-effects. Otherwise, parameter optimization may change program behavior as in Figure 3. As it is not a distinct and unbundled refactoring, we discovered that it is impossible for programmers to manually use an available MtMR in IDEs to construct, say, a Visitor design pattern [19].

Swapping Prefix and Argument Expressions can be addressed by evaluating each prefix/parameter expression of a method call in a distinct statement prior to invoking the method. The following method invocation evaluates expressions \( \exp_1 \) and \( \exp_2 \) and then executes method call \( m \).

\[
\text{expl1.m(exp2);}
\]

2A similar problem occurs when CMSR adds a new parameter whose default value is not a simple variable expression [46]. CMSR in all four IDEs does not check this precondition.
For behavior-preserving MiMR, additional statements are needed to evaluate \( \exp_1, \exp_2 \) and the method call. Assume that expression \( \exp_2 \) below has type \( \text{type2} \):

\[
\text{type1} \ \text{var1} = \exp1; \\
\text{type2} \ \text{var2} = \exp2; \\
\text{var1.m(var2)};
\]

After a move, the order of the \( \exp_1 \) and \( \exp_2 \) evaluations are unchanged:

\[
\text{type1} \ \text{var1} = \exp1; \\
\text{type2} \ \text{var2} = \exp2; \\
\text{var2.m(var1)};
\]

Another approach is always to make MiMR leave a delegate method behind. When a delegate is introduced, the target method signature is not transformed after a move. Consequently, all method invocations are not transformed either as shown in Figure 7.

<table>
<thead>
<tr>
<th>(a) Before</th>
<th>(b) After moving m</th>
</tr>
</thead>
<tbody>
<tr>
<td>class A{}</td>
<td></td>
</tr>
<tr>
<td>\void m(B b) {}</td>
<td></td>
</tr>
<tr>
<td>\void n() {</td>
<td></td>
</tr>
<tr>
<td>\new A().m(new B());</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>class B{}</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 7. Move via Parameter Leaving a Delegate in Move \( A.m(B) \) to \( B.m(A) \)

**Dereferenced Null** detection requires a static null analysis. To date, such analyses are too expensive to be used by incremental compilers, and so too by refactoring engines. Another approach is to extend the Java type system or annotations as existing tools like Checker Framework [47], FindBugs [48] and Java Modeling Language (JML) [49] do. The recent JDT compiler also supports annotation-based null analysis in Java 8. It means that a method whose parameter types are annotated with either \@NonNull \ or \@Nullable \ can detect possible null references at compile-time, making the program free from Null Pointer Exceptions at run-time after a move. We suggest this problem be delegated to the Java compiler community, and leverage what Java itself offers natively. This solution will take time to realize, but ultimately will be correct and dependable.

**Duplicated Prefix Expression** can be addressed by using the approaches described in Swapping Prefix and Argument Expressions above. Suppose we move \( m \) below via field variable \( \text{fld} \) in \( \text{type1} \).

\[
\exp1.m();
\]

We can reuse variable \( \text{var1} \) to hold the prefix value \( \exp1 \) as the value of extra parameter:

\[
\text{type1} \ \text{var1} = \exp1; \\
\text{var1.fld.m(var1)};
\]

Also, use of a delegate method does not produce duplicated prefix expression either (see Figure 8).

| class A{} |
| \void m(B b) {} |
| \void n() { |
| \new A().m(); |
| } |
| class B{} |

(a) Before

| class A{} |
| \void m(B b) { //delegate this.b.m(this); } |
| void n() { |
| \new A().m(); |
| } |
| class B{} |

(b) After moving m

Fig. 8. Move via Field and Leaving a Delegate in Move \( A.m(B) \) to \( B.m(A) \)

**V. RELATED WORK**

We already referenced key papers related to our work. There are other points and papers we want to discuss.

It is now over a quarter century since the refactorings were first described [1], [2]. To the best of our knowledge, there is still no commonly known central repository of technical definitions of primitive refactorings, their preconditions, and code transformation rules; only vague, informal, and underscriptive descriptions exist [50], [51]. The best reference for used preconditions are comments in the source code of the Eclipse refactoring engine [22], for which we are grateful. Still, as a community, we should aspire for more.

Even individual refactorings, such as MiMR, are not particularly well documented. (An exception [52] exists.) Only Tsantalis et al. [11] described ten preconditions of MiMR. We found that eight of them are a subset of the preconditions in Table I. The other two are not required.

Ouni et al. [53] proposed search-based refactorings that minimize changes of program semantics. When program elements are moved among class declarations, the semantic proximity of two classes is measured by evaluating (1) vocabulary similarity in declaration and variable names and (2) dependency of method calls in call graphs and shared field access. It is based on the assumption that a refactored program is syntactically correct and behavior-preserving.

Yang et al. [54] proposed purity-guided refactoring that checks semantic behavior preservation of “pure methods” which are free from side-effects. They showed the approach works for Memoization refactoring that returns the cached result for the same input instead of executing a method call. It requires parameters to have primitive types. They also observed that 22~24% methods in four Java applications satisfy purity.

Finally, we agree with Brant and Steimann with their critiques of refactorings [55]: the existing refactoring (which should be called “transformation”) tools can be immensely helpful in particular cases where correctness is less considered.
VI. CONCLUSIONS

Refactorings are widely understood to preserve program behavior. In our experience, too often they do not. The reasons are clear: not all preconditions are implemented or they approximate what should be used. In surveying four Java IDEs (Eclipse JDT, IntelliJ IDEA, Apache NetBeans, and Oracle JDeveloper), we discovered that all implement different sets of preconditions for MiMR, move-instance-method refactorings. Not surprising, it is not difficult to find programs where each IDE refactoring engine fails to preserve behavior. Moreover, even if the ‘right’ preconditions are used, the code transformations that realize the refactoring may not be behavior-preserving. We also suggested how these flaws could be corrected.

Although Eclipse JDT may have the most reliable refactoring engine, we have reported 25 MiMR-related JDT bugs since 2012 [56]. 15 of them change program behavior and the other 10 produce compilation errors. Only 6 have been fixed to date, and one bug took 266 days to fix [57]. The oldest MiMR bug that we reported has remained unfixed for five years. Other IDEs are no different. We published a technical report [58] on this work in July, 2016. We made very few changes to Table I in January, 2019.

Refactoring is a centerpiece of modern software development. Refactoring engines are standard tools in today’s IDEs, yet their reliability does not hold up to a close scrutiny, as we have shown. They are definitely usable, but their reliability is limited. We see improving refactoring engines as a significant, interesting and intellectual challenge, because the behavior preservation property of refactoring engines is within reach – provided that there are pioneers to make it so.

Acknowledgments. We gratefully acknowledge support for this work by NSF grant CCF-1212683.

REFERENCES