Move-Instance-Method Refactorings:
Experience, Issues, and Solutions

Jongwook Kim
University of Texas at Austin
Austin, TX 78712, USA
jongwook@cs.utexas.edu

Don Batory
University of Texas at Austin
Austin, TX 78712, USA
batory@cs.utexas.edu

Milos Gligoric
University of Texas at Austin
Austin, TX 78712, USA
gligoric@utexas.edu

ABSTRACT
Refactorings, by definition, preserve the behavior of a target program. Such a strong property is encoded by precisely defined preconditions for each refactoring. Only if all preconditions are satisfied will a target program be transformed. Code transformation is done by following the rules defined to produce syntactically-correct, refactored code. Consequently, most behavior-changing violations in refactorings are induced by incorrect preconditions or lack of required checks. In this paper, we show how the transformation rules commonly used in four Java IDEs change program behavior in a Move-Instance-Method Refactoring (MiMr). We report that all have flaws in code transformation and propose solutions for each identified problem in MiMrs.

1. INTRODUCTION
Work on software quality improvement that uses refactorings [23, 38, 39], such as identifying refactoring opportunities [4–8, 34, 43, 45, 50–52], assumes refactorings preserve program behavior. Using off-the-shelf IDE refactorings is presumed to facilitate this work but doing so implicitly adopts frailties of IDE refactoring engines. Refactorings are believed to be built on rock-solid foundations; to our chagrin, this is not the case [22, 47].

Among primitive refactorings, a Move-Instance-Method Refactoring (MiMr) affects many different types of program elements by applying code transformations on access modifiers, method signatures, method calls, and the method’s body. It is the most commonly used and representative refactoring of today’s IDEs [35, 53].

In the last few years, we extensively used the JDT refactoring engine [30] and (as a consequence of the problems that we found) have built a new refactoring engine for Java [31]. In doing so, we have become painfully aware of the limitations of IDE refactoring engines, as well as appreciative of the difficulties to meet the high standards expected of refactoring engines, namely behavior preservation.

In this paper, we distill our experiences for others to appreciate. We investigated how MiMr transforms code in the latest versions of Eclipse JDT [18], IntelliJ IDEA [26], NetBeans [37], and Oracle JDeveloper [40] IDEs. Regardless of preconditions that these Java IDEs check, we found all of them may change program behavior due to incorrect code transformation rules.

2. 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 shows typical examples where method \( m \) is moved from class \( A \) to \( B \) via a parameter and a field variable, respectively.

<table>
<thead>
<tr>
<th>(a) Before</th>
<th>(b) After moving ( m )</th>
</tr>
</thead>
</table>
| class A{
  int i =0;
  void m(B b){
    i++;
  }
  void n(){
    new A().m(new A());
  }
}
| class A{
  int i =0;
  void n(){
    new B().m(new A());
  }
}
| class B{
  void m(A a){
    a.i++;}
}
| class B{
  void m()
}

Figure 1: MiMr via a Parameter

<table>
<thead>
<tr>
<th>(a) Before</th>
<th>(b) After moving ( m )</th>
</tr>
</thead>
</table>
| class A{
  int i =0;
  B b= new B();
  void m(){
    i++;
  }
  void n(){
    new A().b.m(new A());
  }
}
| class A{
  int i =0;
  B b= new B();
  void m(){
    new A().b.m(new A());
  }
}
| class B{
  void m(A a){
    a.i++;}
}
| class B{
  void m()
}

Figure 2: MiMr via a Field

Table 1 lists 19 preconditions used by the Eclipse JDT for MiMr (without leaving behind a delegate method) and
<table>
<thead>
<tr>
<th>Precondition</th>
<th>Eclipse JDT</th>
<th>IntelliJ IDEA</th>
<th>NetBeans</th>
<th>Oracle JDeveloper</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Abstract method</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2. Annotation declaring type</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>3. Conflicting parameter name</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>4. Conflicting target method</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>5. Constructor</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>6. Destination-type variable in the LHS assignment</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>7. Duplicated generic type</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>8. Generic-type destination</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>9. Inaccessible to references</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>10. Interface declaring type</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>11. Native method</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>12. null value of the destination-type parameter</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>13. Polymorphic target method</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>14. Recursive invocation</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>15. Reference to enclosing instance</td>
<td>✔</td>
<td>×</td>
<td>×</td>
<td>✔</td>
</tr>
<tr>
<td>16. Reference to non-local generic type</td>
<td>✔</td>
<td>✔</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>17. Super reference</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>×</td>
</tr>
<tr>
<td>18. Synchronized method</td>
<td>✔</td>
<td>×</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>19. Unavailable destination-type</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

- ✔ precondition checked by IDE. It does not necessarily mean that a correct precondition is used or the precondition is implemented correctly.
- × precondition not checked by IDE.

Table 1: Move-Instance-Method Precondition Comparison.

![Figure 3](image-url)  
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, its elimination changes program behavior [12]. To be fair, checking for side-effects is a complicated and expensive analysis [1, 2, 25, 32, 42, 49].

Surprisingly, JDeveloper does not transform method invocations of a moved method. So method m in Figure 3a is unchanged after the move.

3.2 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 parameter of the original class type. Also, the value of the parameter used as destination in a method invocation becomes the prefix expression after move. By doing so, in Figure 4, two class creation expressions `new A()` and `new B()` are swapped after a move.

![Figure 4](image-url)  
Figure 4: Swapping Prefix & Parameter Expressions

3.1 Parameter Optimization

When an instance method is moved, an extra parameter is added to reference members of the original class declaration. We found that all four IDEs apply parameter optimization [30] in MiMr. It means that the extra parameter is not added when the target method’s body does not reference members of the original class.

```java
class A{
    int i = 0;
    void m(B b, C c, D d){
        i++;
    }
    void n(){
        new A().m(new B());
    }
}
class B{
}

(a) Before

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

(b) After moving m
```

3. CODE TRANSFORM FLAWS

![Figure 3](image-url)  
(a) Before

![Figure 4](image-url)  
(b) After moving m

Figure 3: Parameter Optimization

Figure 4: Swapping Prefix & Parameter Expressions

3 The same problem occurs when Change-Method-Signature Refactoring (CmSr) removes an existing parameter, thereby removing the corresponding expression in a method call [14]. CmSr in all four IDEs fail to check this precondition.
In JDT’s MiMr, the new parameter of the original class type takes the position where the destination-type parameter existed before the move. It 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:

(1) void m(C c, D d, A a) { a.i=0; }

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 order. Therefore, swapping the prefix and parameter expressions changes their evaluation order, which may change program behavior if there are side-effects [13].

NetBeans has a serious bug that always uses this keyword to reference the original 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().

(2) void n(){ new B().m(new C(), new D(), this); }

3.3 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. Only JDT and IDEA check this precondition #12 in Table 1.

3.4 Duplicated Prefix Expression

When a method is moved via a field variable, a different code transformation rule is applied as shown in Figure 2. In a reference to the moved method, the prefix expression is not exchanged with the destination parameter’s expression (Section 3.2). Instead, a field variable name used as the destination is added to the prefix. Also, a new parameter of the original class type is introduced as we described in Section 3.1. 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 [11] as shown in Figure 6. Even when a field variable is used as a destination, NetBeans still transforms method invocations incorrectly as in (3). It drops original prefix new A() and uses this keyword to reference the original class members.

(3) void n(){ b.m(this); }

JDeveloper is also incorrect, as in (4).

(4) void n(){ new A().m(); }

4. PROPOSED SOLUTIONS

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

Parameter Optimization, as implemented in IDEs, is a failure of separation of concerns. Parameter optimization is a distinct refactoring that is bundled with MiMr, 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 MiMr in IDEs to construct, say, a Visitor design pattern [30].

Swapping Prefix and Argument Expressions can be addressed by three different approaches. The first is the most radical, but also the most interesting. Our idea is to create editable views of programs. That is, a view of an object-oriented program P is a series of class/method/field renames, class/method/field moves that are applied to P to produce a semantically equivalent program P’ that is presented to a user to view, edit, execute and debug.

This is the idea behind a new refactoring engine for Java, R3 [31], which has a tiny code base of 5K Java LOC and runs 10x faster than the JDT refactoring engine. Views are possible because R3 does not rely on a program transformation system to manipulate Abstract Syntax Trees (ASTs). Instead, R3 integrates pretty-printing program ASTs with a database of facts; R3 never modifies ASTs. The R3 database
has, for example, a tuple for each method declaration in the program and a tuple for each class. The name of a method is an attribute of that method’s tuple; the same for the name of a class. The class in which a method belongs is an attribute of that method’s tuple pointing to the tuple of the class. R3 refactorings update the database (and never modify program ASTs). Rename-Method simply updates the name field of the method’s tuple; a MiMr updates the method’s tuple to point to the tuple of the class in which it is now a member.

When the AST of a method is displayed, the method’s name is retrieved from the database, and pretty-printed. When a class is displayed, its AST (up to the AST of its members) is displayed by retrieving the class name from the database and printing it. Then all member tuples that belong to that class are retrieved and their AST’s are pretty-printed, yielding a display of a refactored class.

The use of pretty-printing is evident in the following example. Suppose method A.m in (5) has two parameters of types B and C.

\( \text{(5) class } A \{ \text{ void m(B b, C c) } \} \)

The same method expressed as C-language function would have all parameters of m explicit as in (6). Now suppose we declare m as a member of class B in Java. (7) would be its signature: the B-type parameter becomes this and is otherwise implicit. A pretty-print of the method’s AST accomplishes this text translation. If m were declared in class C, (8) would be its signature, where the C-type parameter is this. If m belongs to class D, not matching an existing parameter type, m becomes a static method in D, as in (9), where no implicit this exists in m’s body. Again, pretty-printing accomplishes this text translation.

\( \text{(6) m(A a, B b, C c) } \)
\( \text{(7) m(A a, C c) } \)
\( \text{(8) m(A a, B b) } \)
\( \text{(9) D.m(A a, B b, C c) } \)

Given the above, let’s return to the problem at hand, namely: evaluating method arguments in a fixed order. All arguments of a method call are evaluated in a unique order when the method becomes static as in (9). If all methods have a static definition, it is possible to present a view of a program to make methods appear as non-static members of a class. (Which particular class is part of a view definition.) All methods can be given static definitions in the underlying program; a view could present a completely different image of each method. The version of the program that is sent to the Java compiler would have static methods so that no matter which class a method may be assigned, all method arguments would be evaluated in the same order. And if done properly, one could debug views (see [28]).

The idea of presenting editable views of a program has been demonstrated in X15 [28], although taking views to this next level has not yet been done. As we are the authors of X15, we believe this is possible, but will take time to develop a prototype to demonstrate these ideas.

A second approach is to evaluate each prefix/parameter expression of a method call in a distinct statement prior to invoking the method. Suppose that expression e1 below has type T1. The following method invocation evaluates e1 and e2 in this order and then executes method call m.

\[ e1.m(e2); \]

For behavior-preserving MiMr, additional statements are needed to evaluate e1, e2 and the method call:

\[ T1 v1 = e1; \]
\[ T2 v2 = e2; \]
\[ v1.m(v2); \]

After a move, the order of e1 and e2 evaluation is unchanged:

\[ T1 v1 = e1; \]
\[ T2 v2 = e2; \]
\[ v2.m(v1); \]

Of course, this solution may require a change to the refactoring engine or the Java compiler. In our experience, changing the Java language and/or compiler is very unlikely. Source code transformations to add these statements is possible, but very ugly.

A third approach is always 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.

![Figure 7: Move via Parameter Leaving a Delegate](image)

Dereferenced Null detection requires a static null analysis. To date, such an analysis is 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 [10], FindBugs [19] and Java Modeling Language (JML) [27] 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 believe a good approach is to delegate this problem 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 second and third approaches described in Swapping Prefix and Argument Expressions above. Suppose we move m below via field variable f in T1.

\[ e1.m(); \]

We can reuse variable v1 to hold the prefix value e1 as the value of extra parameter:

\[ T1 v1 = e1; \]
\[ v1.f.m(v1); \]
Also, use of a delegate method does not produce duplicated prefix expression either (see Figure 8).

![Figure 8: Move via Field and Leaving a Delegate](image)

(b) After moving \texttt{m}

5. 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 [23, 38]. 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 descriptive definitions exist (e.g., [21, 29]). The best reference for used preconditions are comments in the source code of the Eclipse refactoring engine [18], 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. Only Tsantalis et al. described ten preconditions of MiMr in [51]. We found that eight of them are a subset of the preconditions in Table 1. The other two are not required.

In [41], Ouni et al. 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 that the existing refactoring (which should be called “transformation”) tools can be immensely helpful in particular cases where correctness is less considered. We agree with Brant and Steimann with their critiques of refactorings in [9].

6. 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, and those that are approximate what should be used. In surveying four Java IDEs (Eclipse JDT, IntelliJ IDEA, NetBeans, and 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 alone have reported 25 MiMr-related JDT bugs since 2012. 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. The oldest MiMr bug that we reported has remained unfixed for two and half years.

Refactoring is commonly positioned as a centerpiece of modern software development. Refactoring engines are common 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 anything but stellar. 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.

7. REFERENCES

[28] Jongwook Kim and Don Batory and Danny Dig. Refactoring software product lines (or editing and refactoring views or programs), 2016.
[40] Oracle JDeveloper 12.2.1.0.0. oracle.com/technetwork/developer-tools/jdev/.