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.

```
class A{
  int i = 0;

  void m(B b){
    i++;
  }

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

class B{
}
```

(a) Before

```
class A{
  int i = 0;

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


class B{
  void m(A a){
    a.i++;
  }
}
```

(b) After moving `m`

Fig. 1. MIMR via a Parameter in Move `A`.m(`B`) to `B`.m(`A`)

```
class A{
  int i = 0;

  B b = new B();

  void m(){
    i++;
  }

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

class B{
}
```

(a) Before

```
class A{
  int i = 0;

  B b = new B();

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


class B{
  void m(A a){
    a.i++;
  }
}
```

(b) After moving `m`

Fig. 2. MIMR via a Field in Move `A`.m() via `A.b` to `B`.m(`A`)

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 lastest 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

Precondition	**Eclipse JDT**	**IntelliJ IDEA**	**Apache NetBeans**	**Oracle JDeveloper**
1. Abstract method	✓	✓	✓	✓
2. Annotation declaring type	✓	✗	✓	✓
3. Conflicting parameter name	✓	✗	✓	✗
4. Conflicting target method	✓	✓	✗	✓
5. Constructor	✓	✓	✓	✓
6. Destination-type variable in the LHS of assignment	✗	✗	✗	✗
7. Duplicated generic type	✓	✓	✗	✗
8. Generic-type destination	✓	✓	✓	✓
9. Inaccessible to references	✓	✓	✗	✓
10. Inaccessible to target method	✓	✗	✗	✗
11. Interface declaring type	✗	✗	✓	✓
12. Native method	✓	✗	✗	✗
13. `null` value of the destination-type parameter	✓	✓	✗	✓
14. Polymorphic target method	✓	✓	✓	✗
15. Recursive invocation	✓	✗	✗	✗
16. Reference to enclosing instance	✓	✗	✗	✗
17. Reference to non-local generic type	✓	✗	✗	✓
18. Super reference	✓	✗	✗	✓
19. Synchronized method	✓	✗	✗	✗
20. Unavailable destination-type	✓	✓	✓	✓
Total	**18**	**9**	**8**	**11**

– ✓ precondition checked by IDE. **It does not necessarily mean that a correct precondition is used or the precondition is implemented correctly. For example, IntelliJ IDEA correctly moves an abstract method only when the method is not run-time polymorphic, but inadequately skips to check if the destination class is abstract.**
– ✗ precondition not checked by IDE.

TABLE I
MOVE-INSTANCE-METHOD PRECONDITION COMPARISON.

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.

```
class A{
  void m(B b){}

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

class B{
}
```

(a) Before

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


class B{
  void m(){}
}
```

(b) After moving `m`

Fig. 3. Parameter Optimization in Move A.m(B) to B.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].[1] 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.

```
class A{
  int i = 0;

  void m(B b,C c,D d){
    i++;
  }

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

class B{
}
```

(a) Before

```
class A{
  int i = 0;

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


class B{
  void m(A a,C c,D d){
    a.i++;
  }
}
```

(b) After moving `m`

Fig. 4. Swapping Prefix & Parameter Expressions in Move `A.m(B,C,D)` to `B.m(A,C,D)`

[1]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 MIMR, 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:

```
(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 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].[2]

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()`.

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

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.

```
class A{
  int i = 0;

  void m(B b){
    i++;
  }

  void n(){
    B b = null;
    m(b);
  }
}

class B{
}
```

(a) Before

```
class A{
  int i = 0;

  void n(){
    B b = null;
    b.m(this);
  }
}


class B{
  void m(A a){
    a.i++;
  }
}
```

(b) After moving `m`

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).[3] 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.

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

```
class A{
  int i = 0;

  B b = new B();

  void m(){
    i++;
  }

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

class B{
}
```

(a) Before

```
class A{
  int i = 0;

  B b = new B();

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


class B{
  void m(A a){
    a.i++;
  }
}
```

(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 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 [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`.

```
exp1.m(exp2);
```

[2]The same problem occurs when CMSR reorders the positions of parameters [37]. This precondition is not checked in all four IDEs.

[3]A 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 $\texttt{exp}_1$, $\texttt{exp}_2$ and the method call. Assume that expression $\texttt{exp}_\texttt{i}$ below has type $\texttt{type}_\texttt{i}$:

```
type1 var1 = exp1;
type2 var2 = exp2;
var1.m(var2);
```

After a move, the order of the $\texttt{exp}_1$ and $\texttt{exp}_2$ evaluations are unchanged:

```
type1 var1 = exp1;
type2 var2 = exp2;
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.

```
class A{
  void m(B b){}

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



class B{
}
```

```
class A{
  void m(B b){    //delegate
    b.m(this);
  }

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

class B{
  void m(A a){}
}
```

(a) Before **(b) After moving** `m`

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 `fld` in $\texttt{type}_1$.

```
exp1.m();
```

We can reuse variable `var1` to hold the prefix value $\texttt{exp}_1$ as the value of extra parameter:

```
type1 var1 = exp1;
var1.fld.m(var1);
```

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

```
class A{
  B b = new B();

  void m(){}

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




class B{
}
```

```
class A{
  B b = new B();

  void m(){    //delegate
    this.b.m(this);
  }

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

class B{
  void m(A a){}
}
```

(a) Before **(b) After moving** `m`

Fig. 8. Move via Field and Leaving a Delegate in Move A.m() 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 undescriptive 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

[1] W. G. Griswold, "Program Restructuring as an Aid to Software Maintenance," Ph.D. dissertation, University of Washington, 1991.

[2] B. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[3] W. F. Opdyke and R. E. Johnson, "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems," in *SOOPA*, 1990.

[4] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, *Recommending Refactoring Operations in Large Software Systems*. RSSE, 2014.

[5] G. Bavota, A. D. Lucia, and R. Oliveto, "Identifying Extract Class Refactoring Opportunities Using Structural and Semantic Cohesion Measures," *Journal of Systems and Software*, Apr. 2011.

[6] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending Move Method Refactorings via Relational Topic Models," *IEEE Transactions on Software Engineering*, Jul. 2014.

[7] G. Bavota, R. Oliveto, A. D. Lucia, G. Antoniol, and Y. Guéhéneuc, "Playing with Refactoring: Identifying Extract Class Opportunities through Game Theory," in *ICSM*, 2010.

[8] V. Sales, R. Terra, L. F. Miranda, and M. T. Valente, "Recommending Move Method Refactorings Using Dependency Sets," in *WCRE*, 2013.

[9] D. Silva, R. Terra, and M. T. Valente, "Recommending Automated Extract Method Refactorings," in *ICPC*, 2014.

[10] R. Terra, M. T. Valente, K. Czarnecki, and R. S. Bigonha, "Recommending Refactorings to Reverse Software Architecture Erosion," in *CSMR*, 2012.

[11] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Transactions on Software Engineering*, May 2009.

[12] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "Recommendation System for Software Refactoring Using Innovization and Interactive Dynamic Optimization," in *ASE*, 2014.

[13] G. Bavota, S. Panichella, N. Tsantalis, M. D. Penta, R. Oliveto, and G. Canfora, "Recommending Refactorings based on Team Co-Maintenance Patterns," in *ASE*, 2014.

[14] N. Tsantalis and A. Chatzigeorgiou, "Identification of Extract Method Refactoring Opportunities for the Decomposition of Methods," *Journal of Systems and Software*, Oct. 2011.

[15] M. Gligoric, F. Behrang, Y. Li, J. Overbey, M. Hafiz, and D. Marinov, "Systematic Testing of Refactoring Engines on Real Software Projects," in *ECOOP*, 2013.

[16] G. Soares, R. Gheyi, and T. Massoni, "Automated Behavioral Testing of Refactoring Engines," *IEEE Transactions on Software Engineering*, Feb. 2013.

[17] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," in *ICSE*, 2009.

[18] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson, "Use, Disuse, and Misuse of Automated Refactorings," in *ICSE*, 2012.

[19] J. Kim, D. Batory, and D. Dig, "Scripting Parametric Refactorings in Java to Retrofit Design Patterns," in *ICSME*, 2015.

[20] K. Wang, C. Zhu, A. Celik, J. Kim, D. Batory, and M. Gligoric, "Towards refactoring-aware regression test selection," in *ICSE*, 2018.

[21] J. Kim, D. Batory, D. Dig, and M. Azanza, "Improving Refactoring Speed by 10X," in *ICSE*, 2016.

[22] "Eclipse Java Development Tools 2018-12 (4.10.0)," https://eclipse.org/jdt/.

[23] "IntelliJ IDEA 2018.3.2," http://jetbrains.com/idea/.

[24] "Apache NetBeans 10.0," https://netbeans.org/.

[25] "Oracle JDeveloper 12.2.1.3.42.170820.0914," https://www.oracle.com/technetwork/developer-tools/jdev/.

[26] M. Schäfer, A. Thies, F. Steimann, and F. Tip, "A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs," *IEEE Transactions on Software Engineering*, Nov. 2012.

[27] F. Steimann and A. Thies, "From Public to Private to Absent: Refactoring Java Programs Under Constrained Accessibility," in *ECOOP*, 2009.

[28] "Eclipse Bug 495899," https://bugs.eclipse.org/bugs/show_bug.cgi?id=495899.

[29] "Eclipse Bug 495902," https://bugs.eclipse.org/bugs/show_bug.cgi?id=495902.

[30] S. Artzi, A. Kieżun, J. Quinonez, and M. D. Ernst, "Parameter Reference Immutability: Formal Definition, Inference Tool, and Comparison," *ASE*, Mar. 2009.

[31] A. Sălcianu and M. Rinard, "Purity and Side Effect Analysis for Java Programs," in *VMCAI*, 2005.

[32] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir, "Transformation for Class Immutability," in *ICSE*, 2011.

[33] J. Quinonez, M. S. Tschantz, and M. D. Ernst, "Inference of Reference Immutability," in *ECOOP*, 2008.

[34] W. Huang and A. Milanova, "ReImInfer: Method Purity Inference for Java," in *FSE*, 2012.

[35] S. Artzi, A. Kiezun, D. Glasser, and M. D. Ernst, "Combined Static and Dynamic Mutability Analysis," in *ASE*, 2007.

[36] "Eclipse Bug 495901," https://bugs.eclipse.org/bugs/show_bug.cgi?id=495901.

[37] "Eclipse Bug 495905," https://bugs.eclipse.org/bugs/show_bug.cgi?id=495905.

[38] "Eclipse Bug 495907," https://bugs.eclipse.org/bugs/show_bug.cgi?id=495907.

[39] N. Ayewah and W. Pugh, "Null Dereference Analysis in Practice," in *PASTE*, 2010.

[40] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," in *PLDI*, 2002.

[41] M. G. Nanda and S. Sinha, "Accurate Interprocedural Null-Dereference Analysis for Java," in *ICSE*, 2009.

[42] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault Localization and Repair for Java Runtime Exceptions," in *ISSTA*, 2009.

[43] R. Madhavan and R. Komondoor, "Null Dereference Verification via Over-approximated Weakest Pre-conditions Analysis," in *OOPSLA*, 2011.

[44] D. Hovemeyer, J. Spacco, and W. Pugh, “Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs,” in *PASTE*, 2005.
[45] “Eclipse Bug 424388,” https://bugs.eclipse.org/bugs/show_bug.cgi?id=424388.
[46] “Eclipse Bug 495903,” https://bugs.eclipse.org/bugs/show_bug.cgi?id=495903.
[47] “Checker Framework,” http://types.cs.washington.edu/checker-framework/.
[48] “FindBugs,” http://findbugs.sourceforge.net/.
[49] “The Java Modeling Language,” http://www.eecs.ucf.edu/~leavens/JML//index.shtml.
[50] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
[51] J. Kerievsky, *Refactoring to Patterns*. Addison-Wesley, 2004.
[52] D. Batory, *Automated Software Design Volume 1*. draft - available from author on request, 2019.
[53] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, “Search-based refactoring: Towards semantics preservation,” in *ICSM*, 2012.
[54] J. Yang, K. Hotta, Y. Higo, and S. Kusumoto, “Towards purity-guided refactoring in Java,” in *ICSME*, 2015.
[55] J. Brant and F. Steimann, “Refactoring Tools Are Trustworthy Enough and Trust Must Be Earned,” *IEEE Software*, 2015.
[56] “JDT Refactoring Bugs,” http://www.cs.utexas.edu/users/schwartz/RR/jdtrefactoringbugs.html.
[57] “Eclipse Bug 416198,” https://bugs.eclipse.org/bugs/show_bug.cgi?id=416198.
[58] J. Kim, D. Batory, and M. Gligoric, “Move-Instance-Method Refactorings: Experience, Issues, and Solutions,” 2016.