Safer Refactorings with Assertions
(Publication Summary)*

Anna Maria Eilertsen
Dept. of Informatics
University of Bergen, Norway
anna.eilertsen@student.uib.no

Anya Helene Bagge
Dept. of Informatics
University of Bergen, Norway
anya@ii.uib.no

Volker Stolz
Dept. of Computing, Mathematics and Physics
Bergen University College, Norway
volker.stolz@hib.no

1 Introduction

Refactoring and restructuring code is a natural part of the development process and can play an important role in software maintenance and evolution. Methodologies like Agile Programming [14] and eXtreme Programming [2] emphasise the importance of continuous refactoring in software development, and small refactorings like *Rename* and *Extract Local Variable* should be performed regularly by the developer, to keep the code clean and avoid technical debt [9, 11].

Refactoring by hand is hard and can introduce errors [17]. To support the ideal practice of frequent refactoring, a number of automated refactoring tools have been developed and common refactorings are often supported by IDEs (integrated development environment). Unfortunately the behavior of automated refactoring tools are not always well documented, and the implementations can differ from what programmers expect. Refactoring preconditions are not always communicated well, neither in the documentation nor in error messages.

Most tools have basic precondition checks in place, with typically at a minimum a syntactical check of the input code. These checks are not necessarily sufficient, and may result in refactorings being applied in cases where preconditions are violated. The reverse can also be true – over-eager preconditions can prevent safe refactorings or refactorings were semantic errors are easily fixed afterwards. The implementation of the refactoring can also be surprising at times, and, as we will soon see in examples, can deviate from what the programmer intended.

Currently, the commonly suggested solution by Fowler, among others, is: *never refactor without a proper test suite*. This correlates with the Fowler's dependence on the developer's own judgement in how to implement the described refactoring in the particular cases. This leads to descriptions that are hard to implement in tools, which in turn leads to ad-hoc implementations that can surprise the programmer, break the code, and introduce subtle changes in the semantics of the code that will not show up as a compilation error, and it requires carefully crafted tests to reveal them. Some of these errors are hard to identify manually, and require that the programmer has a solid understanding of the programming language [20].

Such problems prevents the adoption of tools, and the need for more correct refactoring tools has been voiced several times [3, 8, 12]. At the same time, making the tools too restrictive or rejecting refactoring invocations when the programmer does not understand why it is prevented will also inhibit use, and can make the programmer simply perform the same (possibly unsafe) refactoring by hand [3, 8, 12, 21, 16].

Although proper tests should undeniably be in place, we would like to address the semantic precondition checking. While syntactic preconditions can effectively be checked statically, semantic preconditions can pose

* This paper is based on the first author's master thesis [5], parts of which will be published at ISoLA'16 [6].

Submitted to the 9th Seminar on Advanced Techniques & Tools for Software Evolution, Bergen, Norway, 2016-06

Original code

```
class C {
  X x = new X();

  public void f() {
    x.n();
    m();
    x.n();
  }
  void m(){x = new X();}
}
```

After Extract Local

```
class C {
  X x = new X();

  public void f() {
    X temp = x;
    temp.n();
    m();
    temp.n(); //semantic change
  }
  void m(){x = new X();}
}
```

Listing 1: Example of a behavior-changing application of *Extract Local Variable*. Extracted expression is highlighted in original code and replaced throughout the whole scope. Due to an assignment in another scope this behavior-changing refactoring is allowed by two of the most common Java IDEs, Eclipse and NetBeans.

problems in languages like Java where doing determining whether a path is executed can be undecidable by static analysis [1].

The question is then, *Can refactorings be made safer by encoding preconditions as dynamic checks?* Below, we consider one semantic precondition for *Extract Local Variable* that is hard to check statically and we propose a dynamic check as an alternative. More specifically we aim to encode in the source code a semantic property of a program structure that predicts changed behavior after refactoring, and we introduce a runtime check in the code that will alert the programmer if the structure does not have this property.

In our work [5, 6], summarised below, we illustrate how this, together with a well-covering (although not necessarily complete) test suite can check correctness of two refactorings in Java: *Extract Local Variable* and the composite refactoring *Extract And Move Method* [13]. We validate that our idea would "work in the wild" with a case study, in which we automatically apply a high number of the enhanced refactorings to a commonly used, open-source software project and check the introduced preconditions dynamically to see if they detect changed behavior. This is a considerable step in the direction of a safer industrial strength refactoring tool, that currently has a safer *Extract Local Variable* refactoring ready-for-use with a safer *Extract And Move Method* close behind.

2 Motivating Example

We will give a motivating example of the problem we try to tackle, and show our solution.

Consider the code fragment in Listing 1: a method is repeatedly invoked on the field `x`, which is assigned to between the two method invocations. Our example is simplistic, but in a large code base such behavior may not be as evident, and the structure of our example can be generalized to longer sequences of statements [5]. APIs frequently require sequences of invocations, and to avoid repetition a programmer may decide to refactor this sequence into a new method in the target class or extract a complex, repeated expression into a local variable. In order to assure behavior preservation in this case, the tool must check that the expression is not assigned to in the refactored code.

If the assignment is a statement in the refactored method body, this refactoring would be prevented by most tools, but checking a precondition stating that an assignment cannot happen in any code reachable from the method is hard, and often impossible, in Java. To avoid posing too restrictive preconditions, like preventing the refactoring if an assignment cannot be ruled out, this refactoring will be executed, producing well-formed code with no compile errors, but with a subtle change in semantics: whereas the calls before have been on distinct objects, they are now on a single object. This may or may not be a problem, depending on how the program behavior is defined. It does however, mean that the refactoring can change program behavior without notifying the programmer, which is undesirable in a tool. The same effect can also be observed with the refactoring *Move Method* [5].

In the popular Java IDEs Eclipse and NetBeans, performing *Move Method* on the original example provided in Listing 1 will also yield this kind of behavior change, while IntelliJ will use another implementation of the refactoring with its own problems.

We claim that the crucial behavior change happens at the point in the code where the reference to x is replaced with a reference to *temp*, which happens under the assumption that the two are behaviorally equivalent, and we

propose an encoding of this assumption: `x == temp`. We can check this encoded property dynamically using the `assert` keyword in Java. Adding generation of this dynamic check to the *Extract Local Variable* refactoring will produce the code shown in Listing 2.

Refactored source

```
1  class C {
2    public X x = new X();
3
4    public void f() {
5      X temp = x;
6      temp.n();
7      m();
8      assert x == temp: "Extract Local changed semantics";
9      temp.n(); //semantic change
10   }
11   void m(){x = new X();}
```

Listing 2: The result of applying *Extract Local Variable* with dynamic checks to the code in Listing 1: dynamic checks are added and will alert the programmer at runtime that the objects change.

3 A Condition for Safe Refactoring

Our results show [5] that this property – that *the extracted expression must evaluate to the same value at all its relevant occurences* – is a necessary precondition for both *Extract Local Variable* and *Extract And Move Method* (a particular instance of *Move Method*). For *Extract Local Variable* this will act as an *postcondition*, i.e. a condition that must be true after the refactoring if it was applied correctly; while for *Move Method* and *Extract And Move Method* they can either be used in the same way, or as runtime preconditions. If the code shown in Listing 2 runs without the asserts alerting the programmer, then the same precondition holds for *Extract And Move Method* and *Move Method*. The programmer can apply *Inline Local Variable*, remove the assert (ideally automatically by the tool) and safely apply the more complex refactoring.

In order to run the checks the code containing the asserts must be executed, either by manually running the program or by running a test suite guaranteed to execute these lines of code. Note that the test suite does not need to actually test the result of running the program, and not even to exhaust the execution path, as long as it *ensures coverage of all execution paths that can lead to the refactored code*. This reduces the effort of writing the tests significantly.

4 Plug-In and Case Study

In order to validate our idea that assertions can uncover semantic changes in refactorings, we developed the *Safer Refactoring Plug-in* for Eclipse, with specialised *Extract Local Variable* and *Extract and Move Method* that inserts dynamic checks. Our plug-in is based on an earlier experiment [13]. The plug-in itself is available from `git://git.uio.no/ifi-stolz-refaktor.git`.

For our case study, we wished to verify if the program structures that can trigger semantic changes exist in real-life code and can be refactored with resulting behavior changes. We also evaluate whether the dynamic checks we propose are useful and capture such changes. To validate our idea we execute a large number of our refactorings with dynamic checks on "real" code and see if the introduced asserts will alert us about any behavior change.

We use a heuristic to apply the refactorings, an automated tool for applying them with integrated dynamic checks, and run this from the Eclipse plug-in. When running our experiment we are interested in:

1. can we introduce dynamic tests without breaking the code?
2. do the tests trigger any of the generated assertions in the refactored code?
3. are the triggered asserts *sound*, i.e. do they tell us about actual behavior changes resulting from the refactorings?
4. are the triggered asserts *complete*, i.e. are there behavior changes that are not captured by them (but by the tests)?

We applied the heuristic and the refactorings to the Eclipse JDT UI Project – we believe that this project is a good representative of professionally written Java source code, where the number of contributors contributes to its validity as object of a case study. It comprises over 300.000 lines of code (excluding blanks and comments), with more than 25.000 methods, and has an extensive set of unit tests. The test suite we used is the Automated Test suite for the Eclipse JDT UI project, which we ran using the JUnit workbench in the Eclipse IDE. It is a completely automated test suite, containing 2396 test cases. We also tried the Apache Commons Math Library as input, but that produced very few applications of the refactorings (300 applications of Extract Local Variable), and the only conclusion we were able to draw is that the code style of this project was poorly fitted for our use.

Invoking the project-wide *Extract Local Variable* refactoring on the full Eclipse JDT UI project resulted in 4 538 single refactorings and 7 665 assertions. The refactoring introduced no compile errors. We then ran the Eclipse JDT UI Automated Test suite on the refactored code and found 4 failures, all of which were violations of our generated asserts. In addition we had 133 violations of the generated asserts that were reported in the console output from the tests, but did not seem to affect the test results. Running the test suite without asserts produced no failures and no errors.

The reported assertion violations originated from two specific asserts – in both cases the extracted expression was a get-method that returned newly constructed values each time it was called. Calling such a method twice will produce objects that may evaluate to equal using the `equal`-method, but will not not be reference-equal as checked with `==`, hence breaking our assertions. Thus the triggered assertions likely do not indicate a significant semantic change.

5 Discussion and Conclusion

We are aware that the corner cases are very particular, and that our assessment of them can be biased by our position as refactoring tool developers [17]. This is why we aim to verify that they do indeed exist in a "wild" source code, by implementing an automated tool that can apply the refactorings with checks to a large code base. However, we do agree with Steinmann's stance, that bugs in refactoring tools should not be excused but removed [3].

Extract Local Variable is a commonly implemented and used refactoring, and thus we think is important to guarantee its correctness. It is implemented in IDEs of several languages, and indeed the behavior change we look at can be introduced by a similar operation in other, object-oriented languages.

Behavior change. Calling something a bug in a refactoring invites a discussion on the definition of behavior and refactoring specifications. We aim to not take too much of a stand in regard to behavior and motivate our asserts in the code exactly like this. Without agreement on a definition of behavior, the refactoring tool cannot check if behavior is preserved. Thus its responsibility should be considered to effectively communicate to the programmer what possible behavior changes could have happened, and while this is currently done by the use of previews [8], we believe our assert statements have the advantage of being a runnable contribution to an existing test suite and a possibly lasting change in the code. The latter means that it could serve as a documentation step of which refactoring had been applied to which arguments and what semantic *risks* it introduced. Adding to the test cases by instrumenting the code also means that we can inspect not only the observable behavior, but also the inner program structures. Soares et al. [20] have generated test cases for finding transformations that introduced behavior change, but we see our white-box testing (a test of the internal structure of software) as an improvement.

Specification of Refactorings We are concerned by the lack of refactoring specifications. Both Opdyke [18] and Fowler [9] have attempted refactoring catalogues, but neither can currently serve that purpose. Schäfer et al. [19] give a concise, formal definition of some refactorings that they can translate easily into code for the JastAdd [7] attribute grammar framework for Java. For the refactorings they look at, they are mostly concerned with visibility and shadowing, and consequently make use of infrastructure that tracks such references and either keeps bindings consistent, or rejects a refactoring if the refactored program would have different bindings. Graph transformations have also been used to specify refactorings by Mens et al. [15]; however, in the particular case of the Move Method refactoring, they have opted to only deal with static methods/calls. Also Ó Cinnéide's "minitransformations" preserve behavior due to a restriction to structural manipulation [4].

A challenge in specifying refactorings for common languages is the low "refactorability" of the languages. [3]. Java is a complex, but commonly used language and is a challenge when it comes to defining refactorings. One could try to approach a refactoring catalogue for Java by starting with a subset of the language (like Featherweight Java [10]), try to define refactorings for this ideal subset, and then generalize them for the rest of the language.

The objection is of course that the generalizing can be very hard.

Conclusion

Can refactorings be made safer by encoding preconditions as dynamic checks? We show that, yes, we have cases that can only be checked dynamically and we can insert assertions that perform those checks. But, our initial experiments indicate that there seem to be few cases where such assertions are actually triggered, and further investigation is needed before such an approach is useful to developers.

References

[1] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, Sept 2008.

[2] K. Beck. *Extreme programming explained: Embrace change*. Addison-Wesley Professional, 2000.

[3] J. Brant and F. Steimann. Refactoring tools are trustworthy enough and trust must be earned. *IEEE Software*, 32:80–83, 2015.

[4] M. Ó. Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *International Conference on Software Maintenance, ICSM 1999*, page 463. IEEE Computer Society, 1999.

[5] A. M. Eilertsen. Making software refactorings safer. Master's thesis, Dept. of Informatics, University of Bergen, June 2016.

[6] A. M. Eilertsen, A. H. Bagge, and V. Stolz. Safer refactorings. In *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'16)*, LNCS. Springer, Oct 2016. To appear.

[7] T. Ekman and G. Hedin. The JastAdd system – modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, 2007.

[8] S. Erb. A survey of software refactoring tools. Technical report, Baden-Württemberg Cooperative State University, Karlsruhe, Germany, May 2010.

[9] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[10] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[11] J. Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

[12] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012)*, page 50. ACM, 2012.

[13] E. Kristiansen. Automated composition of refactorings. Master's thesis, Dept. of Informatics, University of Oslo, 2014. Available from `http://www.mn.uio.no/ifi/english/research/groups/pma/completedmasters/2014/kristiansen/`.

[14] R. C. Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.

[15] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software & Systems Modeling*, 6(3):269–285, 2007.

[16] E. Murphy-Hill. Improving refactoring with alternate program views. Technical report, Portland State University, Portland, OR, 2006.

[17] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *Software Engineering, IEEE Transactions on*, 38(1):5–18, 2012.

[18] W. F. Opdyke. Refactoring object-oriented frameworks. Technical Report GAX93-05645, University of Illinois at Urbana-Champaign, 1992.

[19] M. Schäfer and O. de Moor. Specifying and implementing refactorings. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '10*, pages 286–301. ACM, 2010.

[20] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *IEEE Software*, 27(4):52–57, 2010.

[21] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *34th International Conference on Software Engineering (ICSE 2012)*, pages 233–243. IEEE, 2012.