Safer Refactorings

Anna Maria Eilertsen[1], Anya Helene Bagge[1], and Volker Stolz[2(✉)]

[1] Institute for Informatikk, Universitetet i Bergen, Bergen, Norway
anna.eilertsen@student.uib.no, anya@ii.uib.no
[2] Institute for Data- og Realfag, Høgskolen i Bergen, Bergen, Norway
volker.stolz@hib.no

Abstract. Refactorings often require semantic correctness conditions that amount to software model checking. However, IDEs such as Eclipse's Java Development Tools implement far simpler checks on the structure of the code. This leads to the phenomenon that a seemingly innocuous refactoring can change the behaviour of the program. In this paper we demonstrate our technique of introducing runtime checks for two particular refactorings for the Java programming language: Extract And Move Method, and Extract Local Variable. These checks can, in combination with unit tests, detect changed behaviour and allow identification of which specific refactoring step introduced the deviant behaviour.

1 Introduction

Programmers refactor their code frequently [13]. According to Fowler, a *refactoring* is *"a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"* [5]. Refactoring is traditionally done on the source code, and can modify the structure on various levels.

For example, we may increase reuse or readability in almost all programming languages by splitting a large method or function into several or replacing a reoccurring expression with a local variable. Two of the refactorings we will discuss, a variant of Extract Method and Extract Local Variable, are frequently used in many languages.

In object-oriented programs, manipulation of the class hierarchy can also be a refactoring: examples of this are introducing a subclass, or collapsing a subclass into a superclass by repeatedly applying the Pull-up Method/Field refactoring. The other refactoring discussed in this work is the Move Method refactoring. It moves a method, not within the class hierarchy, but rather "side-ways" into a different class. Many of these refactoring steps correlate with software quality metrics, such as number of lines, and number of methods. The Move Method refactoring, for example, can affect coupling between classes.

Refactorings have traditionally been described in the form of patterns. For object-oriented languages these patterns usually consist of a structural

This article is based upon work from COST Action ARVI IC1402, supported by COST (European Cooperation in Science and Technology).

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part I, LNCS 9952, pp. 517–531, 2016.
DOI: 10.1007/978-3-319-47166-2_36

match, and a description of the behaviour of the code. Informal natural language descriptions of behaviour have also been used to "specify" design patterns [6]: ideally programmers match their mental model of the code they are going to write against the available patterns. Various attempts have been made to formalise refactorings, see for example Opdyke's work [14], and Schäfer and de Moor [16], or design patterns, as in Cinnéide [2].

Describing the behaviour of refactorings, either formally or informally, pose various challenges. Natural language may impose ambiguities in the descriptions, while formal specifications are hard to communicate. Firstly, while the required static structure of software can be described concisely, through e.g. a class diagram, there is no agreed-on, commonly used notation for behaviour. Secondly, even though refactorings can be formalised for ideal subsets of programming languages (like Featherweight Java [8]), the resulting specification is not easily generalised to the full language. Currently, implementations in industrial-grade refactoring tools must be ad-hoc, and may consequently introduce subtle semantic changes. This is the case in e.g. Eclipse and NetBeans. In fact, an inspection of the Eclipse bug tracker reveals numerous cases of refactorings producing code that no longer compiles correctly.

As we will soon see in an example, an ad-hoc refactoring may accidentally change the behaviour of a program. According to Fowler's commonly used definition of refactorings above, this must be interpreted as a bug, and the "refactoring" should not have been applied in the first place. While it is relatively straightforward to check for structural issues, such as overriding a method, there can also be more subtle changes in the heap at run time. The behaviour might in fact change without any compiler warning, and the developer must rely on having suitable unit tests to uncover the newly introduced undesired behaviour.

Consider the fragment of code in Listing 1. Since the code in class `C` uses the public field `x` to call methods repeatedly, and since the field is not declared final, at runtime the value of `x` changes between the two method invocations.

```
1  public class C {
2    public X x = new X();
3
4    public void f() {
5      x.m(this);
6      // Not the same x.
7      x.n();
8    }
9  }
```

```
public class X {
  public void m(C c) {
    c.x = new X();
    // If m is called from
    // c, then c.x no longer
    // equals 'this'.
  }
  public void n() {...}
}
```

Listing 1: Source and destination for Extract- and Move-Method refactoring, before refactoring.

From informal observations we conclude that developers do not expect such intermittent reassigments as in our example, but work on the assumption that attribute values in syntactic proximity do not change. They expect the simpler behaviour of calling the methods on the same object for adjacent lines of code. Our example is certainly simplistic, but in a large code base such behaviour may

not be evident. The pattern in our example can be generalised to longer (not necessarily contiguous) sequences of statements, using navigation path expressions with varying prefixes.

APIs frequently require sequences of invocations, and best practices of programming require to avoid repetition: as a programmer is invoking methods on the same variable, she may decide to refactor this sequence into a new method in the target class (assuming that the source code for the target is under her control, and not e.g. in a library). In case this variable is a local variable already, extracting and moving the statements is safe, as long as the local variable is not reassigned. If it is a non-final attribute though, as we have seen, the refactoring will produce valid code, but with changed behaviour: whereas the calls before have been on distinct objects, they are now on a single object, obviously giving the program a different meaning.

As this problem in general cannot be detected statically at the time the refactoring is applied, we combine the refactoring with the generation of an assertion that will report at runtime if the refactoring had been incorrectly applied. Applying the refactoring with the generated assertion on the above example, we obtain the following:

Refactored source

```
1 public class C {
2   public X x = new X();
3
4   public void f() {
5     x.h(this);
6   }
7 }
```

New method h

```
public class X {
  public void h(C arg) {
    m(arg);
    /* generated by our approach: */
    assert arg.x == this;
    n();
  }
}
```

Listing 2: Changed code after refactoring.

We claim that it is easy for developers to make this mistake in practice: the illustrated refactoring step can easily be applied in e.g. Eclipse or IntelliJ through the Move Method refactoring, possibly preceded by the Extract Method refactoring. There are no checks that will warn the programmer about the changed behaviour. We note that a similar effect can be observed when extracting to a local variable: should any side-effect manipulate the value of the extracted sub-expression, the original code will execute subsequent method calls on different objects, and the refactored code on a single object. Both patterns are very similar, and we will see that our approach covers both.

Proposed Solution. The changed behaviour can be easily detected at runtime, if we *encode the necessary assumptions into assertions*. For the above example, it is straightforward to first store the target of the method call in a new variable, passing it along, and checking for object equality in the newly introduced method body, see again Listing 2.

We present the following contributions: (1) our technique of generating assertions for the Extract-Local and Move-Method refactorings, (2) a drop-in

replacement for the Extract-Local refactoring with assertion generation for the Eclipse JDT, and assertion generation for the Extract-And-Move-Method refactoring plugin we developed in earlier work [10].

2 The Refactorings

In this section, we describe the two refactorings, how our assertions capture the semantic requirements, and the underlying theory.

Fowler's "observable behaviour" is open to interpretation up to a certain degree, even the notion that a refactored program should show the same input/output behaviour as the original code: are differences in intermediate output tolerated, e.g. when restructured control flow leads to different debugging output? In the absence of method specifications, does this notion apply to method output (results), or only for the cases covered by unit tests?

Here, we take the position that for an object-oriented program, the observable behaviour can also be understood as a *sequence of method invocations on particular objects* during the execution of a program. Note that this opens the possibility that we consider the refactoring as incorrect (different sequence), even though the output is unchanged. It is easy to see that the Extract-And-Move refactoring above produces different execution histories, as will the Extract-Local refactoring. Without going into the depth of the argument, we also observe that some refactorings require a notion of *history refinement*: the Extract Method refactoring preserves existing calls in the history, but adds intermediate calls to the newly created method. Extract Local Variable can collapse multiple calls to the same method into a single one. Other structural manipulations such as Pull-Up Method that modify the inheritance hierarchy, may, if incorrectly applied, preserve the objects in the history, but lead to calling different virtual methods with the same name.

2.1 Extract Local

Extract Local Variable (also called Extract Variable, or Introduce Explaining Variable [5, p. 124]) is a pattern for replacing a repeating expression with a reference to a local variable initialised to said expression.

Extract Local Variable takes as input an expression e and a consecutive selection of statements S. It declares a variable v and initialises it to the value of e. Then all occurrences of e in S are substituted with a reference to v.

The problem with respect to behaviour-preservation appears if e evaluates to different values throughout the selection, i.e. we are making a substitution where the introduced variable does not have the same value at that point as the original expression. The problem appears because v will be fixed to the value e evaluates to at that line, regardless of whether the expression e would evaluate to different values in the original expression where it has been replaced by v.

The underlying problem is essentially the requirement for a precise *points-to analysis* [15,17] : we need to know that for all statements in the selected range, the target expression e evaluates always exactly to the same object.

Optimizing compilers or JVMs that would like to minimise redundant loads of fields, i.e. heap accesses where the value cannot have changed in between, use this analysis and suffer from the same limitations. The optimization in the compiler is known as Common Subexpression Elimination, and done statically and conservatively. On the JVM-level this has been tackled under the name "Hot Field-analysis" by Wimmer and Mössenböck [19] through an aggressive, dynamic technique that efficiently switches back to the unoptimized and correct behaviour, in case it detects that the relevant heap has been modified.

Thus, to ensure correctness of Extract Local Variable we check at every substitution that the introduced variable evaluates to the same value as the replaced expression, i.e. `assert v == e`. In some cases we can guarantee correctness without asserts under any of the following conditions:

- e is only referred to once in the program
- e is a local variable and it is not assigned to in S
- all segments of e are field references with the `final` modifier

The expression argument could also contain method calls, which introduces yet another problem concerning side-effects: if a method has side effects, it matters how many times it is called. In our work we assume e (but not S) to be free of side effects, and we do not pursue this problem further.

Before Extract Local Variable

```
1
2
3   public void f() {
4     a.b.c.d.m();
5     a.b.c.d.n();
6     a.b.foo(a.b.c.d);
7     a.b.bar();
8     a.b.c.d.m();
9     }
```

After Safer Extract Local Variable

```
public void f() {
  X temp = a.b.c.d;
  temp.m();
  assert temp == a.b.c.d;
  temp.n();
  assert temp == a.b.c.d;
  a.b.foo(temp);
  a.b.bar();
  assert temp == a.b.c.d;
  temp.m();
}
```

Listing 3: Safer Extract Local Variable

Our Safer Extract Local Variable includes the following algorithm for inserting asserts, to be performed after the original refactoring is finished:

Let e be the expression argument, S a contiguous selection of statements, and v be the newly introduced variable:

for each statement s in S:

if s contains an expression with subexpression v

insert the following statement in S, before s: `assert v == e;`

We illustrate this with the example in Listing 3.

The Extract-Local refactoring can be applied to expressions of any non-void type, whether it is an object, or a primitive value. In the following, we will take this refactoring as the starting point for a more complex, object-oriented refactoring, where we will be only interested in object references.

2.2 Extract and Move

Generalising from the example in Listings 1 and 2, it is easy to see why a developer would want to Extract and Move such a fragment: from a software-quality metrics perspective, the so-called coupling between the two involved classes can be decreased. However, as our example illustrates, the resulting behaviour is not as clear-cut as it may seem at the first glance.

While the extraction of code fragments into a new temporary method (with a fresh name) is unproblematic, the origin of the problem lies in the updates to the navigations paths upon moving the temporary method into the new type. In fact, the Move Method-refactoring demonstrated here, as implemented for example by the Eclipse refactoring tools for Java, is not the only interpretation of what said refactoring should do.

Opdyke's original idea

```
class X {
  public void h(C c) {
    c.x.m();
    // Not necessarily the same x.
    c.x.n();
} }
```

Our idea

```
class X {
  public void h(C c) {
    m();
    assert c.x == this;
    n();
} }
```

Listing 4: Result of Opdyke's refactoring (left) compared to ours (right)

Alternative characterizations of Move Method-refactorings (and a formalization) for C++ programs was given by Opdyke in his seminal PhD thesis [14, Sects.8.5 and 8.6]. He offers two alternatives, one where a reference back to the original object is passed as a parameter, and another where those references are handled through an additional field in the destination class. The snippets in Listing 4 contrasts the outcome of Opdyke's refactoring that uses parameters with our refactoring. The method is assumed to have been moved into the declared type for the field $c.x$. We observe that Opdyke's solution with parameters is more general, as it also preserves behaviour in the case of heap-manipulation, since it preserves all field accesses from the original code. The alternative solution with a field access requires a program flow analysis in the precondition. Both of his solutions yield more complex navigation paths and increased coupling, whereas in our case, coupling can be reduced.

Our approach requires passing additional data that is used in the equality check in the assertions. It serves the purpose of a *ghost variable* (see [7] for a discussion of their usefulness and a critique), and should be understood as existing on the level of a specification: after discharging the assertion (and thus proving correctness of the refactoring step), the variable could be removed.

While in the case of Extract Local, this is only an additional local variable, in the case of Extract And Move Method, this results in an additional parameter that gets passed into the newly extract method. Care must be taken in the subsequent development steps that no other dependencies are introduced on this variable, so that when the assertions eventually get removed/discharged,

the unused parameter can also be removed, and thus avoiding increased coupling between classes. Nonetheless, other subexpressions used in the extracted method may also need to be passed as additional parameters, and increase coupling—but these are the responsibility of the refactorer.

Original code

```
class C{
  public void f() {
    x.m(this);
    x.n();
} }
```

Extract Local w/assert

```
class C{
  public void f() {
    X temp = x;
    temp.m(this);
    assert temp == x;
    temp.n();
} }
```

Extract Method

```
class C{
 public void f() {
    X temp = x;
    h(temp);
  }
  public void h(X temp){
    temp.m(this);
    assert temp == x;
    temp.n();
} }
```

Move Method

```
class C{
  public void f() {
    X temp = x;
    temp.h(this);
} }
class X{
  public void h(C c){
    this.m(c);
    assert this == c.x;
    this.n();
} }
```

Listing 5: Our Extract And Move Method with asserts starts with applying Extract Local Variable, which also introduces the asserts. Next, we extract the method, and use the extracted variable as the target for Move Method. Finally, the extracted variable `temp` can be safely inlined again (not shown).

Conceptually, Extract And Move Method is composed by Extract Method and Move Method. In our safer version we also need to introduce assertions to check that the value of the target expression does not change throughout the selection, i.e. method call. These are the same assertions as for Extract Local Variable, where we check that the value of the extracted variable does not change throughout the selection. Thus, we perform Extract And Move Method by composing Extract Local Variable, with Extract Method, followed by Move Method. A final Inline Variable removes the extra variable introduced by Extract Local Variable. This is illustrated in Listing 5. Extract Method and Move Method are defined as follows.

Extract Method [5, p. 110] takes as input a consecutive selection of statements S occurring in a class C. It introduces a new method m in C, with S as the method body. All occurrences of S in C are replaced with a call to m. Arguments are restricted in that they must form a syntactically correct method body with one return type or void. If S refers to local variables declared before S, these are passed as arguments to the method. If one local variable, v, that is assigned

to in S is used in subsequent code, then the assigned value will be the return value of m, and v will be assigned the result of the method call. If two or more local variables assigned to in S are used in subsequent code, then S is not a legal selection. We will call a selection fulfilling these properties *well-formed*.

Move Method [5, p. 142] takes as input a method m in a class C and an expression e. The expression argument can only contain one or more segments of field lookups or local variables and cannot contain any method calls or other operators than the dot-operator. It declares a new method n in the type of e. If S refers to members of C, n will have an extra parameter c of type C. The method body of n is S with all occurrences of `this` replaced with c and all occurrences of e replaced with `this`. m is then removed from the original class, and all calls to m are replaced with a similar call to n. The argument given to c is `this`.

The safer Extract And Move Method takes two arguments: the expression argument e, corresponding to the target of Move Method, and the selection of statements S, corresponding to the selection argument to Extract Method. These are passed to Extract Local Variable, resulting in a local variable v. Then S is extracted to a new method m, taking at least one parameter corresponding to v. We then move m to the target type of e; a new parameter c is introduced, and all references to the v-parameter is replaced by the `this`-keyword. Obsolete parameters are removed. Remaining references to members in C is now referenced through c. Finally, in the original class, we inline all occurrences of v.

3 Experiment

In the previous section, we have described in detail and applied in examples our refactoring, and have shown that a violating example can be created easily. To validate our idea, that this semantic change could happen in the wild, and that our asserts would capture it, we did a case study. Our *case* is a large code base, representative for object oriented code. We decided to use the Java programming language. Since the asserts are runtime checks, we needed the code to actually run. Thus we focused on finding a code base with a well-covering test suite. We would run our refactoring on the code, then run the tests, and see:

1. do the tests trigger any of the generated assertions in the refactored code?
2. are the triggered asserts *sound*, i.e. do they tell us about actual behaviour changes resulting from the refactorings?
3. are the triggered asserts *complete*, i.e. are there behaviour changes that are not captured by them (but by the tests)?

To do this we needed an implementation of our refactorings that could be automatically applied to a large code base, which required finding sensible arguments for the refactorings: where to apply, and which target expression to use.

We developed a tool for automatically applying both refactorings in appropriate places across an entire Java code base. Our tool contains a heuristic for where a developer would think to apply the refactorings, and executes both refactorings with generated asserts. The heuristic for Extract and Move Method was

partially developed in earlier works [10] with the intention of reducing coupling between classes. We have adapted the Extract And Move heuristic as described below, and developed a similar heuristic for Extract Local Variable. The heuristic finds suitable arguments for each refactoring, including the "target" type for Extract And Move Method. What we have previously referred to as the expression argument of the refactorings, is picked from a set of possible arguments called *prefixes*. A prefix is a qualifier, field access, local variable or `this` keyword, and the heuristic enumerates the set of possible prefixes. For Extract Local Variable we extend the notion of prefix to include single-line getters: a method whose name starts with "get" and whose method body contains a single return statement. This is the only method call we allow as expression. In general we cannot know which method is called at runtime, so we look up the method in the static type of its qualifier. For Extract And Move Method we did not include getters, or any other kind of methods. The heuristic excludes prefixes where the selection contains a statement assigning a new value to the prefix, a variable declared in the selection, local type, unmodifiable type, etc. [9, 2.7].

We extended both heuristics to exclude cases where we knew the asserts would hold, or where we knew the code would break, as explained below. For Extract Local Variable almost all prefixes are allowed, but if the expression has only one segment *and* only occurs once, then there is no need to extract it into a local variable. For Extract And Move Method there are more exlusions:

- If e has only one segment *and* occurs only once, by the same reasoning we exclude it from our prefixes.
- If e is a local variable, it cannot be changed by a method call, and the refactoring will not contribute to our findings.
- If e is a field, it has to be visible from the resulting method n, otherwise we cannot generate syntactically correct assertions. This can be remedied by generating getters, but we did not pursue this idea yet.
- If e's type has generic type arguments, we exclude it, as it is not trivial to move a method into such a class.
- If e is a member of an anonymous type we will not be able to access n, and we exclude such prefixes.
- If e is a static class that will make n static, which is usually undesirable, and we exclude such prefixes.

After pruning the set of prefixes, we rank the remaining expressions after number of occurrences and number of segments. The top ranked expression will be considered the best candidate to the refactoring. The best selection argument will be the selection containing the best candidate.

In the following, we report on our results of combining the two ideas: automated, search-based refactoring and assertion generation.

Firstly, for a case study, we identify a convincingly large, non-trivial Java project that is amenable to our analysis and transformation. Secondly, this project has to have a reasonable amount of existing unit tests that we can run after the transformation to see whether assertions are triggered.

We chose the Eclipse JDT UI project. We believe that it is a good representative of professionally written Java source code with many contributors over the years. It comprises over 300.000 lines of code (excluding blanks and comments), with more than 25.000 methods, and comes with an extensive set of unit tests.

Experiment Implementation

We implemented our refactorings in a plugin for Eclipse. Our plugin supports an interactive and an automated search-based version of both refactorings. They can be invoked either on a method or a project. The interactive Extract Local refactoring can also be invoked directly on a well-formed selection of statements. Invoking a search-based refactoring on a method causes our heuristic to analyze the method to find suitable arguments for the refactoring. Next, our program will execute the refactoring on the candidate provided by the heuristics or the user. Here we are heavily supported by Eclipse's implementation of the refactorings Extract Local Variable, Extract Method and Move Method.

In an Eclipse-instance with our plugin, we imported the Eclipse JDT UI code for version 4.5 (with all dependencies) and the corresponding tests. Before the refactorings were invoked on the code, we ran the Automated Test Suite, where all unit tests passed. The test code was not refactored. We invoked the Extract Local Variable refactoring on the whole project, and ran the tests on the resulting code. We then invoked the Extract And Move Method refactoring on the original code, and ran the tests on the resulting code. We did not refactor already refactored code.

Invoking the *Search-Based Extract Local Variable* refactoring on the full Eclipse project resulted in 4.538 single refactorings and 7.665 assertions. The results are summarised in Table 1. The refactoring introduced no compile errors. We then ran the Eclipse JDT UI Automated Test suite on the refactored code. The test suite finished with 4 failures and 11 errors. The difference between a failure and error in this case, is whether the test expected an exception or error, or not. The 4 failures originated from violation of our generated asserts. The 11 errors were due to build issues, where the build file required an old version of

Table 1. These are the results of our experiment

	Extract and move method	Extract local variable
Executed refactorings	755	4538
Generated asserts	610	7665
Resulting compile errors	14	0
Tests failing before	0	0
Tests errors	84	11
Tests failures	161	4
Asserts triggered in tests	0	2
Instances of asserts triggered	0	137

Java that did not handle our generated asserts, and consequently one file did not finish building. Changing the target Java version in the build file resolved the build problem and removed these 11 errors. 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 (after modifying said build file).

The reported assertion violations originated from two specific asserts. In both cases the extracted expression was a get-method. In one case it seemed to be a factory-method. In the other case the assert was triggered by a method returning a fresh string, where the string object is created in the getter instead of accessing a field or otherwise stored reference. Calling such a method twice will produce objects that may be object-equal (depending on the `equal`-function), but will not be reference-equal (as checked with `==`).

We invoked the *Search-Based Extract And Move Method* refactoring on the full (unrefactored) Eclipse project, resulting in 755 applied refactorings and 610 assertions. This produced 14 compilation errors. Initially we had 180 compile errors, and we incrementally improved our heuristic to exclude targets that would introduce the different types of errors, as previously explained. 3 of the 14 compile errors were due to project specific settings (e.g. an error on unused import). Most compile errors were due to references to enclosing instance, reference to non-visible or unaccessible members, and missing imports. Running the Automated Test suite on the resulting code (with compile errors) produced 84 errors (test not completed due to compilation errors) and 161 failures (unit test not having the expected result). No asserts were found violated. Manually correcting all compile errors (as good as we could) and rerunning the tests produced no errors or failures, and still no assertion violations. Thus, we did not sift through the original test errors and failures with the intention of cataloguing their source.

We should point out that for Extract And Move Method we still had some refactorings that were executed but without generated asserts. Our tool aborted the insertion of asserts if it was clear (usually due to visibility issues) that the asserts would produce a syntactically incorrect program. We did not keep a history of the method-level changes in the refactoring, and did not undo the ones where the algorithm found it impossible to generate asserts. This means that we are only applying the runtime check at a fraction of our Extract And Move Method refactorings. In future work, we would like to introduce special get-methods for these cases. Another approach would be to increase visibility of fields, but this would require yet another check of correctness.

Threats to validity. The following issues have to be kept in mind when considering the experimental results:

- The number of identified instances where the Extract Method refactoring can be applied depends on the quality of the code base. A "perfectly refactored" project, or a project using less object-orientation, will have a lower number of possible instances.
- As described above, we had applications of Extract And Move Method where we could not generate assertions due to issues of field-visibility. This lowers

the potential for assertions to be triggered (although changed results could still be uncovered by failing unit tests).
– Our evaluation uses unit tests to detect changed behaviour. Our results depend on the coverage of the test suite.
– The total number of executed Extract And Move Method refactorings with generated asserts is rather low. We may need a much higher number of applied refactorings to find a violating instance.

We conclude that our experiment was suited for finding the results we needed, and we would like to repeat it with an improved version of the refactoring tool for more code bases. The implementation of the assert generation is not yet ideal, as these results tell us. Nonetheless, the results are promising and there are many improvements that can be done.

4 Conclusion and Future Work

Our research is motivated by the observation that common refactorings can easily, and accidentally, change a program's behaviour. We have presented our idea of improved refactorings, where their semantic correctness conditions are encoded as assertions. As these conditions are impossible, or at least difficult, to check statically, we think that runtime checks present a suitable tradeoff. The generated assertions also serve the additional purpose of documenting which refactoring has been applied and what its semantic *risks* are.

The assertions capture the necessary conditions on the heap for the Extract Local and the Extract And Move Method refactoring. While the former is a standard refactoring, we have implemented the latter as a combination of existing refactorings based on earlier work. We have evaluated our approach by refactoring a code base in the same way as we anticipate developers would do. We execute existing unit tests and observe if the generated assertions are triggered, which would indicate that the refactoring indeed changed the behaviour.

Our findings show a limited success, in that some assertions are violated. This means that a developer *may* accidentally apply the refactoring incorrectly. However, our experimental setup yields a low number of applied refactorings and generated assertions. In Future Work below, we discuss how we could collect more empirical evidence as to the usefulness of our assertions.

Related work. Opdyke already gave refactorings a formal treatment and considered behaviour preservation essential [14]. In his variations of the "Moving Members into a Component" refactorings, he carefully gives formal preconditions which make sure that the refactorings are structurally sound. He is aware that behaviour will not be preserved upon intermediate reassignments to members: "[. . .] all references to each moving member will point to the same location at all times. Program flow analysis would be needed to determine this." [14, p. 130]. This is the problem that we try to tackle dynamically here.

Schäfer and de Moor [16] give a concise, formal definition of some refactorings that they can translate easily into code for the JastAdd [4] 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. They do not go as far as e.g. work on refinement, where it is even formally proved using graph transformations that (consistent) renaming preserves the semantics [20]. Graph transformations have also been used to specify refactorings by Mens et al. [12]; however, in the particular case of the Move Method refactoring, they have opted to only deal with static methods/calls, even unlike Opdyke's original solution, where dependencies would at least be passed by additional parameters. Also Ó Cinnéide's "minitransformations" preserve behaviour due to a restriction to structural manipulation [2].

Soares et al. [18] have generated test cases when applying a refactoring to uncover non-behaviour preserving transformations. We see our approach as a more fine-grained attempt, that during testing (e.g. through unit tests a la Soares), can inspect the object graph in much more detail than just observing the output of the unit tests.

Future Work. While we have a proof-of-concept with hand-written examples, our larger case study in combination with automated refactorings [10] did not reveal many interesting instances of refactoring-induced problems. In future work, we would therefore like to extend our experiment to larger code bases, and identify deficiencies in other refactorings that could be addressed in a similar way. Additionally, in combination with repository mining, it would be interesting to identify where/when in a repository one of our supported refactorings has been applied, add our assertions, and see if we can discover any changed behaviour. Also, we could have a group of software developers use our refactoring, and observe their experience. As we lack the capacity for either set up, we have opted for a more automated solution.

The attentive reader may have noticed that it is not necessary to run the fully refactored code to detect if the Extract And Move Method refactoring changed the behaviour. It is sufficient to only generate the assertions, and then use e.g. a test suite to observe if the semantic preconditions hold. Only after all generated assertions have been covered by the execution without revealing a problem, we would actually apply the final step of the refactoring and move the method.

Accessibility and visibility of the prefix argument to the refactoring is a problem in assertion generation: the Extract And Moved Method refactoring may be applied in more situations than we can generate assertions for. We would like to solve this by generating additional getters to the required information that will only be used by our asserts (and hopefully discarded along with them following subsequent advances in proof support for object-oriented programs). This would increase the number of checks per applied refactoring.

An alternative to using Java's assertions is JML [11], which would have the advantage that the assertions would not pollute the source code, and the additional state-keeping would be confined to JML ghost variables. Also, a custom IDE that understands the notion of these variables and parameters, and could

thus hide them and the generated assertions from the human eye, would most likely improve adoption among developers of our approach. Such an IDE could also take care of any other code modifications like the special getters above, that only need to be available intermittently for the purpose of runtime verification of the refactorings, but should not be visible—or accessible—to developers.

More ambitiously, it would also be possible to attempt to discharge the assertions, which would amount to a correctness proof of an instance of the refactoring (as opposed to proving the entire refactoring correct). We have experimented with the KeY theorem prover [1], which has been able to automatically discharge the vacuous assertions in the trivial example program. This could be attempted unattended in the background after applying the refactoring, or extended to involve a *proof engineer*, who, as support to the actual programmers, attempts to discharge the generated assertions.

The Git repository with our Eclipse-based Java refactorings to reproduce our experiment is available at git://git.uio.no/ifi-stolz-refaktor.git. Additional details are published in a Master thesis [3].

References

1. Ahrendt, W., Beckert, B., Hähnle, R., Schmitt, P.H.: KeY: a formal method for object-oriented systems. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 32–43. Springer, Heidelberg (2007)
2. Cinnéide, M.Ó., Nixon, P.: A methodology for the automated introduction of design patterns. In: International Conference on Software Maintenance, ICSM 1999, pp. 463–472. IEEE Computer Society (1999)
3. Eilertsen, A.M.: Making software refactoring safer. Master's thesis, Department of Informatics, University of Bergen (2016)
4. Ekman, T., Hedin, G.: The JastAdd system - modular extensible compiler construction. Sci. Comput. Program. **69**(1–3), 14–26 (2007)
5. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1994)
7. Hofmann, M., Pavlova, M.: Elimination of ghost variables in program logics. In: Barthe, G., Fournet, C. (eds.) TGC 2007. LNCS, vol. 4912, pp. 1–20. Springer, Heidelberg (2008). doi:10.1007/978-3-540-78663-4_1
8. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. (TOPLAS) **23**(3), 396–450 (2001)
9. Kristiansen, E.: Automated composition of refactorings. Master's thesis, Department of Informatics, University of Oslo (2014). http://www.mn.uio.no/ifi/english/research/groups/pma/completedmasters/2014/kristiansen/
10. Kristiansen, E., Stolz, V.: Search-based composed refactorings. In: 27th Norsk Informatikkonferanse, NIK. Bibsys Open Journal Systems, Norway (2014)
11. Leavens, G.T.: JML's rich, inherited specifications for behavioral subtypes. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 2–34. Springer, Heidelberg (2006)

12. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Softw. Syst. Model. **6**(3), 269–285 (2007)
13. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Trans. Softw. Eng. **38**(1), 5–18 (2012)
14. Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992). UMI Order No. GAX93-05645
15. Ryder, B.G.: Dimensions of precision in reference analysis of object-oriented programming languages. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 126–137. Springer, Heidelberg (2003). doi:10.1007/3-540-36579-6_10
16. Schäfer, M., de Moor, O.: Specifying, implementing refactorings. In: Cook, W.R., Clarke, S., Rinard, M.C. (eds.) Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) 2010, pp. 286–301. ACM (2010)
17. Smaragdakis, Y., Balatsouras, G.: Pointer analysis. Found. Trends Program. Lang. **2**(1), 1–69 (2015)
18. Soares, G., Gheyi, R., Serey, D., Massoni, T.: Making program refactoring safer. IEEE Softw. **27**(4), 52–57 (2010)
19. Wimmer, C., Mössenböck, H.: Automatic feedback-directed object inlining in the Java HotSpot$^{\text{tm}}$ virtual machine. In: Krintz, C., Hand, S., Tarditi, D. (eds.) 3rd International Conference on Virtual Execution Environments VEE, pp. 12–21. ACM (2007)
20. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Asp. Comput. **21**(1–2), 103–131 (2009)