Search-based composed refactorings

Erlend Kristiansen[1], Volker Stolz[1,2*]
1: Institutt for informatikk, Universitetet i Oslo, Norge
2: Institutt for data- og realfag, Høgskolen i Bergen, Norge
stolz@ifi.uio.no

Abstract

Refactorings are commonly applied to source code to improve its structure and maintainability. Integrated development environments (IDEs) such as Eclipse or NetBeans offer refactoring support for various programming languages. Usually, the developer makes a particular selection in the source code, and chooses to apply one of the refactorings, which is then executed (with suitable pre-condition checks) by the IDE.

Here, we study how we can reuse two existing refactorings to implement a more complex refactoring, and use heuristics to derive suitable input arguments for the new refactoring. We show that our combination of the Extract Method and Move Method refactoring can automatically improve the code quality on a large Java code base.

1 Introduction

According to Fowler, *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"* [2]. For object-oriented programs, relevant measures of software quality for example include the amount of coupling to other classes (how many "partners" a class needs to achieve its functionality) and depth of navigation paths: the Law of Demeter [5] prescribes that accesses and calls should only be to direct members of an object, and use intermediate methods to traverse the object graph. As we will here concern ourselves with programs written in the Java language, the latter rule can be understood as not using expressions such as "`x.y.z`" or "`a.getB().getC()`", but instead introducing helper methods. Those measures *decrease* the complexity of the methods, at the less important cost of an *increased* number of methods. This leads to path expressions with only a single dot.

In Section 2, we illustrate our "Extract and Move Method"-refactoring, and show how it is composed out of two existing refactorings. We include remarks on the formal correctness of the refactoring. After that, we describe in Section 3 our analysis of Java source code to derive suitable input for our refactoring. In Section 4, we evaluate quality and performance of our approach, based on refactoring the Eclipse JDT source code. Section 5 concludes with related and future work.

*Partly funded by the EU project FP7-610582 (Envisage).
This paper was presented at the NIK-2014 conference; see http://www.nik.no/.

Before

```
1  class C {
2    A a; B b;
3    X x;
4    void m() {
5      x.y.foo();
6      x.y.bar();
7    }
8  }
9
10 class X {
11   Y y;
12   /* ... */
13 }
14
15 class Y {
16   void foo(){ ... }
17   void bar(){ ... }
18 }
```

After

```
1  class C {
2    A a; B b;
3    X x;
4    void m() {
5      x.fooBar();
6    }
7  }
8
9  class X {
10   Y y;
11   /* ... */
12   void fooBar() {
13     y.foo();
14     y.bar();
15   }
16 }
17
18 class Y ... /* unchanged */
```

Listing 1: The CBO of class `C` improves from 4 to 3 through refactoring.

2 The Extract and Move-Method Pattern

First, we illustrate on an example how our *Extract and Move Method* refactoring uses Cockburn's Protected Variations-pattern [11] to improve the design of the code. For this, we will manually derive suitable selections and targets for delegation. We show how the refactoring impacts the relevant software quality metrics. Then, we show how to decompose our refactoring into two phases, which are both already implemented in the Eclipse Java Development Tools.

The example in Listing 1 shows on a very simple Java example the intended workings of our refactoring. On the left-hand side in method `m` in class `C`, we find two consecutive statements that call methods on the same member.

Arguably, the statement sequence would better be in a method of class `X`, thus avoiding repeated indirection—note that we do not expect any performance improvements, but just want to improve readability of the code. The right-hand side of the listing shows the intended outcome: the original statement sequence has been replaced with a single method call to a new method `fooBar` in class `X` of the corresponding member `x`. In the target method, the navigation path has been adjusted correspondingly. Repeated application of the refactoring could then replace the body of `X.fooBar()` with the invocation of another *new* method `Y.fooBar()` where we will then find the two method calls on the current object.

This example also serves to illustrate how the Coupling Between Object Classes (CBO) metric [1] improves in this case. An object is coupled to another object if one of them acts on the other by using methods or instance variables of the other object. This relation goes both ways, so both outgoing and incoming uses are counted. Each coupling relationship is only considered once when measuring CBO for a class.

Before refactoring the class `C`, it has a CBO value of 4. The class uses members of the classes `A` and `B`, which accounts for 2 of the coupling relationships of class `C`. In addition to this, it uses its variable `x` with type `X` and also the methods `foo` and `bar` declared in class `Y`, giving it a total CBO value of 4.

With respect to the CBO metric, the refactoring action accomplishes something important: it eliminates the uses of class `Y` from class `C`. This means that the class `C` is no longer coupled to `Y`, only the classes `A`, `B` and `X`. The CBO value of class `C` is

therefore 3 after the refactoring, while no other class have an increased CBO.

The example shown here is an ideal situation. Coupling is reduced for one class without any increase of coupling for another class. There is also another important point: to reduce the CBO value for a class, we need to remove *all* its uses of another class. This is achieved for the class `C` in Listing 1, where all uses of class `Y` are removed by the *Extract and Move Method* refactoring. However, we will later see that the refactoring may move dependencies from one class to another, possibly resulting in a net increase.

Two phases: extract and move

Our intended refactoring can be accomplished in two phases. In the first phase, we can extract the selected fragment into a new private method within the same class, passing additional context that it relied on (such as local variables) as arguments. In the second phase, we move the method declaration into a more suitable class (see Section 3), and update the target of the call in the place of the original fragment.

Both refactoring steps are already available as individual refactorings in Eclipse. This saves us a cumbersome and certainly error-prone (re)implementation of our desired refactoring. Additionally, this decomposition allows us to better analyse a failing refactoring (i.e., either the result no longer compiles, or even a crash in the refactoring engine).

For the first phase, we use the JDT's `ExtractMethodRefactoring` class. It mainly requires as arguments a compilation unit (the Eclipse-internal representation of Java classes), offset and length within this unit of the code that should be extracted, and the method name to use for the new method in the current class. Currently, we generate names, which consequently lack the semantic hints from user-provided names. In the second phase, the `MoveInstanceMethodProcessor` requires a handle on the method to be moved (straight-forward to obtain and unique, as we just created it in the previous step), and a variable binding from a declaration as target. Both refactorings offer additional options for fine-tuning, e.g. wrt. to visibility, that we elide here.

The life-cycle of both refactorings is similar and is managed by Eclipse's language-independent toolkit LTK [12]: first, relevant pre-conditions are checked, i.e. whether the refactoring can be successfully executed on the given arguments. After that, the actual change is executed. The toolkit also provides other interactive editing support such as undoing/redoing.

Note on correctness

To the attentive reader, it is obvious that the proposed refactoring is *not always correct* in the sense that preserves the original behaviour. Let us consider the following example in Listing 2. Note that the method `m(C c)` of class `X` assigns to the field `x` of the argument `c` that has type `C`.

Before the refactoring, the methods `m` and `n` of class `X` are called on *different objects* (see lines 5 and 7 of the original class `C` in Listing 2). Yet after the refactoring, they are called on the *same object*. The method `f` of class `C` is now calling the method `f` of class `X` (see line 5 of class `C` in Listing 3), and the program now behaves different than before.

Even though we do not have a solution yet to avoid this problem (except only refactoring on final fields and local variables), let us make the following remarks:

```
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 2: Refactoring this example *does not* preserve behaviour.

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

```
1  public class X {
2      ...
3      public void f(C c) {
4          m(c);
5          n();
6      }
7  }
```

Listing 3: Result of applying the refactoring to method `f` from Listing 2.

firstly, analysing this particular situation requires a precise aliasing analysis, which is especially difficult for an object-oriented language with virtual method calls, and for an open system where not all code paths are statically known[7]. Secondly, the problem already exists with manually refactored code. A developer makes a judgement based on inspection of the code. In our automated setting, we just compound the issue through the scale, or number, of refactorings that we apply. We point out a potential solution to this problem in our conclusion in Section 5, where we harness assertions to at least have safeguards after the fact.

3 Searching for the Right Arguments

In the previous section, we hand-picked the sequence of statements that should be extracted, and the target class corresponding to the member where the new method should be created. These two dimensions can usually be easily identified by a programmer for a given piece of code. Here however, our interest is in automatically improving the quality of an entire code base. Therefore, we need an analysis which identifies candidates and executes the refactoring on them. Even though a statement can span several lines, in the following we will equate statements with lines for simplicity. For control structures such as `for`, `while` or `if-then-else`, we also consider their bodies/branches as part of the statement.

Text selections within method bodies

For a sequence of statements, the text selections that can be generated from it, are equal to all its sub-sequences. Listing 4 shows the possible statement ranges that can be derived from the different contiguous chunks of statement sequences in the example. We indicate with shaded boxes the two different nesting levels; the outermost, lighter-coloured level is the entire method body, and the darker boxes indicated nested blocks within control statements.

Each nesting level of a method body can have many such sequences of statements. The outermost nesting level has one such sequence, and each branch contains its own sequence of statements. We additionally list all 23 possible valid text selections, giving their start/end lines. Every regular statement can be its own sub-sequence,

```
1  class C {
2    A a; B b; boolean bool;
3
4    void method(int val) {
5      if (bool) {
6        a.foo();
7        a = new A();
8        a.bar();
9      }
10
11     a.foo();
12     a.bar();
13
14     switch (val) {
15     case 1:
16       b.a.foo();
17       b.a.bar();
18       break;
19     default:
20       a.foo();
21     }
22   }
23 }
```

Level 1 (10 selections):

$\{(5,9),(11),(12),(14,21),(5,11),$
$(5,12),(5,21),(11,12),(11,21),(12,21)\}$

Level 2 (13 selections):

$\{(6),(7),(8),(6,7),(6,8),(7,8),$
$(16),(17),(18),(16,17),(16,18),(17,18),(20)\}$

Prefixes with number of occurrences for selection (5,21):

$\{a(6), b(2), b.a(2)\}$

Unfixes for (5,21):

$\{a\}$

Listing 4: Source code with high-lighted, nested sequences (left), all possible subsequences given with start/stop line-numbers (right) and prefixes/unfixes.

the `break` statements that manipulate the flow of control are ignored.

Identifying a suitable target

"Target" refers to the member that we will move the fragment to, and invoke the method on. A target is chosen from a set of candidates that are calculated from the selected statements. Here, we will only consider local variables or fields as potential targets, although the practice can be generalised to method calls (especially in the case of so-called "train wrecks" of method calls like `getA().getB().m()`).

We collect our set of candidates per selection using a visitor on the abstract syntax tree (see [3] for implementation details). A candidate is also called a *prefix*, because we are interested in the left-most component of navigation expressions. Certain statements render a prefix subsequently unusable. We call the set of unusable prefixes *unfixes*.

A common reason for a candidate to become an unfix is when it occurs on the left-hand side of an *assignment*: in the relocated code, the updated write access would be to the value `this`, which is not legal in Java. Also, any *local variable* declared within a selection cannot be a prefix for the same reason, as the target must be defined before the statements to move.

The second major source of unfixes is the restriction that, to actually apply the move-step of the refactoring, we must be able to *modify the source code of the target class*. Clearly this is not the case for any of Java's standard classes, nor any classes present only in binary form on the class-path. On the right of Listing 4, we show the calculated candidates: the prefixes a, b and $b.a$ occur, but a is also listed in the unfixes due to being the target of an assignment, and unusable for the refactoring.

Among the remaining two candidates, we use the following heuristics to pick the single target for our refactoring: we prefer the prefix that is most frequently referenced within the selection. If two prefixes have equally many occurrences, we take the first segment of the one with the largest number of segments. This favors

indirection, as it may lower coupling between classes. Here in our case, since the prefixes b and $b.a$ overlap, the choice thus resolves to delegating to member b.

Ruling out invalid selections

Certain constructs within a selection automatically disqualify either the selection or a candidate, because the code after the refactoring would be invalid. The following reasons are the most common causes why a selection is not valid input for the refactoring during our case study:

- **Calls to protected or package-private methods.** The new method may need increased visibility which can conflict with restricted visibility in a sub-class.
- **Double class instance creation.** Consider a field `name` of the enclosing class, and the expression `new A(new B(name))`. If this expression is located in a selection that is moved to another class, `name` will be left untouched, instead of being prefixed with a variable of the same type as it is declared in, leading to broken code (due to a bug in the underlying refactoring).
- **Inner class.** For a similar reason, the instantiation will break as during the move the context of the instantiation has to be updated, or is not available.
- **References to enclosing instances of the enclosing class.** Ditto, relocated code will no longer have access to "outer" members.
- **Inconsistent return statements.** When a selection contains a return statement, then every possible execution path within the selection must end in either a `return` or a `throw` statement.
- **Ambiguous return values.** The extracted fragment would have to return more than a single value due to assignment statements to local variables.
- **Other illegal statements/expressions.** Any expression referring to `super`, and statements changing flow of control like `break/continue`.

Due to the richness of the Java language this list is by no means complete. As any oversight may lead to an easily detectable compilation error in the refactored code, it is merely a matter of convenience to avoid those situations. In principle, if a refactoring fails, we can roll back to the original code and try the next candidate. Alas, we have not succeeded in implementing the composed refactoring with rollback-support through the LTK due to technical limitations in the Eclipse-provided frameworks (LTK & Java-specific refactorings). We invite the reader to refer to [3, p. 62] for a detailed description of the technical issues we encountered.

4 Experiment on the Eclipse JDT UI source code

To lend reasonable credibility to our efforts, we show that on the one hand we are able to automatically analyse and transform a substantial software project, and that on the other hand the refactoring improves the design as expected. To this end, we will analyse parts of the Eclipse source code. We have chosen the Eclipse JDT UI project, because we believe that it is a good representative for professionally written Java source, that many people have contributed to over the years. It comprises

Table 1: Statistics for refactoring the Eclipse JDT UI project.

Time used			
Total time	98m38s		
Analysis time	14m41s	(15%)	
Change time	74m20s	(75%)	
Miscellaneous tasks	9m37s	(10%)	
Numbers of each type of entity analyzed			
Packages	110		
Compilation units	2,097		
Types	3,152		
Methods	27,667		
Text selections	591,500		
Numbers for *Extract and Move Method* refactoring candidates			
Methods chosen as candidates	2,552		
Methods NOT chosen as candidates	25,115		
Candidate selections (multiple per method)	36,843		
Refactorings executed	**Total**	*Extract*	*Move*
Attempts	2,552	2,483	2,469
Not fully executed	83	69	14

over 300,000 lines of code (excluding blanks and comments), with more than 25,000 methods. Furthermore, it comes with an extensive set of unit tests that we can use to evaluate if the refactoring changes the behaviour.

Analysis, refactoring, and calculating the software metrics before and after take time, and our resources limit the amount of source code that we can reasonably evaluate our automation on. In addition, we also compare the result of the included unit tests from before and after the refactoring, to make observations about any changes in behaviour. The experiment has been carried out on a recent laptop with a Intel CoreTM i5-3210M processor and 8 GB of RAM. Another smaller experiment of applying the automated refactoring to our own code that implements the refactoring can be found in [3].

Statistics

Table 1 shows the statistics of batch-refactoring the Eclipse JDT UI project from the Eclipse GIT repository, branch `R3_8_maintenance`. 75% of the total time goes into making the actual code changes. The time-consumers are here the primitive *Extract Method* and *Move Method* refactorings. Included in the change time is the parsing and precondition checking done by these refactorings, as well as textual changes done to files on disk. All this parsing and disk access is time-consuming, and constitutes a large part of the change time.

The pure analysis time, which is the time used on finding suitable refactoring candidates, only makes up for 15% of the total time consumed. This includes analyzing almost 600,000 text selections, while the number of attempted executions of the *Extract and Move Method* refactoring is only about 2,500. So the number of executed primitive refactorings is approximately 5,000. Assuming the time used on

Table 2: Results for analyzing the Eclipse JDT UI project, before and after the refactoring, with SonarQube (bold numbers are better.)

Number of issues for each rule	**Before**	**After**
Avoid too complex class	81	**79**
Classes should not be coupled to too many other classes (Single Responsibility Principle)	**1,098**	1,199
Control flow statements should not be nested too deeply	1,375	**1,285**
Methods should not be too complex	1,518	**1,452**
Methods should not have too many lines	3,396	**3,291**
NPath Complexity	348	**329**
Complexity per function	3.6	**3.3**

miscellaneous tasks is used mostly for parsing source code for the analysis, we can say that the time used for analyzing code is at most 25% of the total time. This means that for every primitive refactoring executed, we can analyze about 360 text selections. With an average of about 21 text selections per method, we can analyze over 15 methods in the time it takes to perform a primitive refactoring.

Refactoring candidates

Out of the 27,667 methods that were analyzed, 2,552 methods contained selections that were considered candidates for the *Extract and Move Method* refactoring. This is roughly 9% of the methods in the project. These 9% had on average 14.4 text selections that were considered possible refactoring candidates.

Executed refactorings

2,469 out of 2,552 attempts on executing the *Extract and Move Method* refactoring were successful, giving a success rate of 96.7%. The failure rate of 3.3% stems from situations where the analysis finds a candidate selection, but the change execution fails. This failure could be an exception that was thrown, and the refactoring aborts. It could also be that the precondition checking for one of the primitive refactorings gives us an error status, meaning that if the refactoring proceeds, the code will contain compilation errors afterwards, forcing the composite refactoring to abort.

Out of the 2,552 *Extract Method* refactorings that were attempted executed, 69 of them failed. This gives a failure rate of 2.7% for the primitive refactoring. In comparison, the *Move Method* refactoring had a failure rate of 0.6% of the 2,483 attempts on the refactoring.

Quality metrics

We use the SonarQube analysis tool to calculate well-known software quality metrics before and after the refactoring. As Table 2 shows, as expected, in general complexity decreases, as we create around 2,500 smaller methods. However, we note that coupling has actually increased—contrary to our expectations from Section 2. SonarQube reports an increase from 1,098 to 1,199. Next, we investigate in a concrete case where this deterioration comes from.

```
1   public class ShowActionGroup extends ActionGroup {
2     /* ... */
3     private void initialize(IWorkbenchSite site, boolean isJavaEditor) {
4       fSite= site;
5       ISelectionProvider provider= fSite.getSelectionProvider();
6       ISelection selection= provider.getSelection();
7       fShowInPackagesViewAction.update(selection);
8       if (!isJavaEditor) {
9         provider.addSelectionChangedListener(fShowInPackagesViewAction);
10      }
11    }
12  }
```

Listing 5: Portion of the `ShowActionGroup` class before refactoring.

Listing 5 shows a portion of the class `ShowActionGroup` from the JDT UI project before it is refactored with the search-based *Extract and Move Method* refactoring. Before the refactoring, the `ShowActionGroup` class has 12 outgoing dependencies.

During the benchmark process, the search-based *Extract and Move Method* refactoring extracts the lines 5 to 10 of the code in Listing 5, and moves the new method to the move target, which is the field `fShowInPackagesViewAction` with type `ShowInPackageViewAction`. The result is shown in Listing 6.

```
1   public class ShowActionGroup extends ActionGroup {
2     /* ... */
3     private void initialize(IWorkbenchSite site,
4                             boolean isJavaEditor) {
5       fSite= site;
6       fShowInPackagesViewAction.generated(
7                               this, isJavaEditor);
8     }
9   }
```

```
1   public class ShowInPackageViewAction
2           extends SelectionDispatchAction {
3     /* ... */
4     public void generated(
5         ShowActionGroup showactiongroup, boolean isJavaEditor) {
6       ISelectionProvider provider=
7                   showactiongroup.fSite.getSelectionProvider();
8       ISelection selection= provider.getSelection();
9       update(selection);
10      if (!isJavaEditor) {
11        provider.addSelectionChangedListener(this);
12      }
13    }
14  }
```

Listing 6: `ShowActionGroup` and `ShowInPackageViewAction` after refactoring.

After the refactoring, the `ShowActionGroup` has only 11 outgoing dependencies. It no longer depends on the `ISelection` type. So our refactoring managed to get rid of one dependency, which is exactly what we wanted. The only problem is, that now the `ShowInPackageViewAction` class has got two new dependencies, the types `ISelectionProvider` and the `ISelection`. The bottom line is that we eliminated one dependency, but introduced two more, ending up with an overall program that has more dependencies now than when we started out.

What can happen in many situations where the *Extract and Move Method* refactoring is performed, is that the *Move Method* refactoring "drags" with it references to classes that are unknown to the method destination. In those situations where a destination class does not know about the originating class of a moved method, the *Move Method* refactoring most certainly will introduce a dependency. This is because there is a bug[1] in the underlying move-refactoring, making it pass an instance of the originating class as a reference to the moved method, regardless of whether the reference is used in the method body or not.

We also note the suboptimal decision in Listing 6 to pass `this` as an argument and use an indirection (which could be eliminated by another application of the pattern) to access the field. Instead, the original formal argument `site` to the

[1]Eclipse Bug 228635 - [move method] unnecessary reference to source

method could have been used. This example thus also illustrates how sensitive the refactorings are to developers' coding styles, as the refactorings are syntax-driven. We cannot expect the refactoring logic to make complex decisions about aliasing—while using the parameter here will be correct, note that in general, especially for concurrently executed code, this decision may not be correct. In our opinion, programmers should prefer to use local variables or parameters instead of fields whenever possible, to enable exactly this type of refactorings. We conclude in remarking that with a better choice of parameter, the complexity of the refactored code would be lower, as one import less would be required.

Unit-test results and technical limitations

Alas, when running the automated refactoring on the code base, in many cases the refactoring results in code that does not compile. This usually hints at missing precondition checks either on our side, or on the side of the two component refactorings. Arguably, refactorings should avoid producing broken code. Although we have tried to avoid most of the problems created by compilation errors or crashes by extending our precondition checks, there is still a substantial amount of errors in the generated code for reasons that we still need to identify. As a consequence, we have only been able to re-run those unit tests that refer to correctly compiled code.

Of the 5,822 test cases in the two provided unit test suites, we have thus only been able to re-run 3,001 due to compilation errors in the refactored code. In those tests, we found 2 new failures after refactoring, i.e., a unit test that passed before refactoring, and not producing an expected result afterwards. We have inspected the implicated code paths, but have not been able to ascertain how the refactoring affected the outcome, due to our lack of domain knowledge for this particular code.

5 Conclusion and Related Work

We have presented a composite refactoring that extracts and moves statement sequences from method bodies, with the goal to improve software quality metrics by following the Protected Variations pattern. The refactoring can be applied automatically to Java code in the Eclipse integrated development environment.

The experiment on part of the Eclipse source code revealed deficiencies in the code that we rely on: the two individual refactorings for extraction and moving occasionally throw exceptions or produce code that does not compile for certain inputs. We try to mitigate this by checking preconditions particular to the observed errors, and have reported three of them in the Eclipse bug tracker (see again [3]).

The choice of the "Extract and Move Method" refactoring was based on observations in our own prototypical software development, where we frequently found code benefitting from the refactoring. As Larman points out [4, p. 431], the improvements decrease maintenance effort in the early stages of coding, but may not be necessary in the presence of stable structural connections between classes.

We observed in Listing 4 that the per-method complexity decreases (as we are splitting methods), yet coupling reported by the metrics tool per class did not improve. We have shown that this is because of the required additional imports in the target class for the refactoring. Here, a more fine-grained metric which averages coupling *per method* could be developed that would show the benefit more clearly. We have also not yet investigated by how much repeated application of the

refactoring improves the result. Instead of running our experiment on the arguably mature Eclipse code base, we might expect to see a larger number of candidates and improvements on source code in earlier stages of development.

The current heuristics of where to apply a refactoring is quite simple. In the future, we would like to improve the analysis phase to identify more valuable selections and candidates (those that yield bigger improvements in quality). This is important, as although analysis is fast, executing a refactoring is the expensive step, which (together with the inability to efficiently roll back) precludes speculative execution to try different possibilities in the search for ideal candidates: at the moment, we take a peep-hole view on statement sequences in individual methods. It is straight-forward to construct an example where two seemingly equally suitable candidates for a selection exist, yet the overall coupling will be differently affected in each case due to the content of the target class.

Further future work includes creating assertions to ensure behaviour preservation. As pointed out in Section 2, our refactoring may currently change behaviour in the presence of aliasing. Even though through assertions we cannot ensure correct behaviour, at least they can notify us during unit tests (assuming sufficient coverage). The assertions can even serve as documentation within the source code for which refactoring step introduced the erroneous behaviour. Even more ambitiously, developers might then set out and discharge these assertions with a theorem prover.

Within the field of refactoring, the opinion about whether refactorings should preserve behaviour is divided. While here we argue for the correctness of our refactoring (and point out its limitations), proponents of the more relaxed view point out that any erroneous behaviour should be captured through unit tests eventually. As Fowler puts it: "If you want to refactor, the essential precondition is having solid tests." [2, p. 73]. In fact, Soares et al. [9], derive unit tests based on the applied refactoring that is likely to uncover changes in behaviour.

A bottom-up approach is taken by Schäfer et al. [8] where they give a concise, formal definition of some refactorings that they can translate easily and correctly into code for the JastAdd attribute grammar framework for Java.

Automatic refactoring of code has been tried out with success at scale by Google to migrate away from deprecated APIs [13].

O'Keeffe et al. [6] present an empirical study of different algorithmic approaches to search-based refactoring. These algorithms generate a set of changes to a program and then use a fitness function to evaluate if they improve its design or not. The fitness function consists of a weighted sum of different object-oriented metrics, whereas we are mostly guided by the CBO metric. The refactorings which they applied however are on the structural level of classes (e.g. moving fields and methods), and not the statement level.

Offering compositional refactorings instead of monolithic, wizard-based interfaces as they are provided in the common IDEs has been the focus of a study by Vakilian et al. [10]. These refactorings shall be accessed via keyboard shortcuts or quick-assist menus and be promptly executed. The mechanism provides developers with an option of performing small rapid changes instead of large changes with a lesser degree of control. The authors hope this will lead to fewer unsuccessful refactorings.

Our work covers common Java language constructs up to Java 7, and could be applied to related object-oriented languages. The upcoming language extensions in Java 8 will require additional scrutiny both from the perspective of the argument

analysis, as will the existing refactoring code have to be updated.

All technical detail is available from [3], as is the Eclipse plugin that implements the analysis and refactoring (both as binary and as source code).

References

[1] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

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

[3] 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/`.

[4] C. Larman. *Applying UML and Patterns*. Prentice Hall, 3rd edition, 2005.

[5] K. J. Lieberherr, I. M. Holland, and A. J. Riel. Object-oriented programming: An objective sense of style. In N. K. Meyrowitz, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 323–334. ACM Press, 1988.

[6] M. O'Keeffe and M. Ó. Cinnéide. Search-based refactoring: An empirical study. *Journal of Software Maintenance and Evolution*, 20(5):345–364, 2008.

[7] B. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In G. Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 126–137. Springer-Verlag, 2003.

[8] 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.

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

[10] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, and R. E. Johnson. A compositional paradigm of automating refactorings. In G. Castagna, editor, *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 527–551. Springer-Verlag, 2013.

[11] J. Vlissides, J. O. Coplien, and N. L. Kerth. *Pattern Languages of Program Design*, volume 2. Addison-Wesley, 1996.

[12] T. Widmer. Unleashing the power of refactoring. `https://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/`, Feb 2007. Last accessed July 2014.

[13] H. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan. Large-scale automated refactoring using ClangMR. In *Proc. of the 29th Intl. Conf. on Software Maintenance*, pages 548–551. IEEE, 2013.