Stepping Stones over the Refactoring Rubicon
Lightweight Language Extensions to Easily Realise Refactorings

Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor

Programming Tools Group, University of Oxford, UK
{max.schaefer,mathieu.verbaere,torbjorn,oege}@comlab.ox.ac.uk

Abstract. Refactoring tools allow the programmer to pretend they are working with a richer language where the behaviour of a program is automatically preserved during restructuring. In this paper we show that this metaphor of an extended language yields a very general and useful implementation technique for refactorings: a refactoring is implemented by embedding the source program into an extended language on which the refactoring operations are easier to perform, and then translating the refactored program back into the original language. Using the well-known *Extract Method* refactoring as an example, we show that this approach allows a very fine-grained decomposition of the overall refactoring into a series of micro-refactorings that can be understood, implemented, and tested independently. We thus can easily write implementations of complex refactorings that rival and even outperform industrial strength refactoring tools in terms of correctness, but are much shorter and easier to understand.

1 Introduction

According to its classic definition, refactoring is the process of improving the design of existing code by behaviour-preserving program transformations, themselves called refactorings. Applying refactorings by hand is error-prone since even very simple operations such as renaming a program entity can affect large parts of the refactored program and may interact with existing program structure in subtle ways, leading to uncompilable code or, even worse, to code that still compiles but behaves differently.

For many years now, popular Integrated Development Environments such as JetBrains' IntelliJ IDEA [10], Eclipse [2], or Sun's NetBeans [17] have provided support for automated refactorings in which the user specifies the refactoring operation to perform, and the refactoring engine performs the requested transformation while checking that program behaviour is actually preserved.

Traditionally [19], this is done by checking preconditions that are thought sufficient to preserve some invariants, which in turn ensure behaviour preservation. However, as we have pointed out before [21], this approach has severe weaknesses. Deriving the correct preconditions relies on a global understanding of the object language in which programs to be refactored are written, and has to account for all corner cases that might possibly lead to an incorrect refactoring. In a complex modern language like Java this is an arduous task even for very simple refactorings, and we found in an informal survey that none of the most popular IDEs solves it satisfactorily [3]. Even if sufficient preconditions are found, further evolution of the language is likely to introduce new

S. Drossopoulou (Ed.): ECOOP 2009, LNCS 5653, pp. 369–393, 2009.
© Springer-Verlag Berlin Heidelberg 2009

constructs and concepts that impact refactoring in subtle ways, and make it necessary to revise the preconditions.

We believe that it is more beneficial to think about refactorings directly in terms of invariant preservation rather than preconditions. The ultimate invariant to be preserved is still, of course, the semantics of the input program, but this criterion is too deep and too far removed from the program as a syntactic entity to be useful. We should rather search for more syntactic invariants whose preservation we can see as ensuring the correctness of the refactoring. In previous work, we have shown how this approach can be used to tackle the *Rename* refactoring: From the abstract criterion of binding preservation, i.e. the requirement that in the renamed program every name still refer to the same declaration as before, we obtain a concrete implementation that correctly refactors many programs on which industrial strength tools fail.

In this paper, we follow this line of work and show how it can be extended to cover more complex refactorings, among them the *Extract Method* refactoring, that was dubbed "Refactoring's Rubicon" by Fowler [7], who proclaimed it to be the yardstick of "serious" refactoring tools.

Our key idea is to treat refactorings not as transformations on programs in the object language that the programmer writes them in, but instead as transformations on programs translated into a richer language that offers some additional language features to ease their implementation. The implementation of the refactoring itself then becomes much simpler, but some effort has to be invested into the translation back from the enriched language to the base language.

As an example, first consider renaming. If we extend our basic language with *bound* names, i.e. names that do not follow the normal lookup rules, but directly bind to their target declaration (preventing accidental shadowing or capture), renaming becomes trivial to implement: First, all names in the input program are replaced by their bound equivalents (their bindings are "locked"). Now the renaming can be performed without having to worry about altering the binding structure, since all references in the program are fixed. Finally, we need to go back to the language without bound names, replacing them with possibly qualified names in the base language that have the same binding behaviour (their bindings are "unlocked"). If unlocking cannot be performed, the transformation is unsafe and has to be aborted and rolled back.

Of course, the unlocking step is highly non-trivial to perform and hard to implement, but a general translation from "Java with Bound Names" to plain Java is very useful in the context of other refactorings as well. Consider, for example, the *Push Down Method* refactoring, in which a method m is moved from a class A to its subclass B. Its crucial correctness property is again binding preservation, since we want to ensure that all calls to m still resolve to the right method after pushing, and that all references to fields, variables, types, and methods inside m itself still refer to the same targets as before. Again, this is easily achieved in Java with Bound Names: we simply lock all calls to m and all names within m itself, then move the definition of m from A to B, and unlock, using the same translation from Java with Bound Names to plain Java originally developed for *Rename*.

In this paper we show how this idea of refactorings as transformations on a richer language can be extended to more complex refactorings, most prominently *Extract*

Method, *Inline Method*, *Extract Local Variable*, and *Inline Local Variable*. Although binding preservation is still important, these refactorings are more challenging in that they move code relative to other code. As a criterion to preserve behaviour, we suggest control flow and data flow preservation: All statements in the affected methods should maintain their control flow predecessors and successors throughout the refactoring, and all variables should have the same reaching definitions.

Our aim is not to prove that this is a sufficient criterion for behaviour preservation. If we wanted to do that, we would have to restrict our attention to a suitably well-behaved subset of the language, since in full Java even the introduction of a number of extra `push` instructions that would be needed to realise the call to an extracted method in byte code, could possibly lead to an out-of-memory error, which would alter the behaviour of the program.

We rather take flow preservation as a common-sense criterion that is to guide our implementation; since it can be effectively checked, it also provides a safety net for our refactoring engine. By judicious introduction of additional language extensions we can further decompose the overall transformation into micro-refactorings that each perform a small, well-defined task. These language extensions are lightweight in that they cannot occur in source programs and no code is generated for them, but they are introduced and eliminated again during the process of refactoring.

This decomposition brings with it the usual benefits of modularity, as it eases implementation and testing . We were thus able to implement all four of the above-mentioned refactorings in less than 3000 lines of code, most of which is reused heavily between the individual refactorings, as part of our JastAdd-based refactoring engine, and pass all applicable tests in the internal test suites of both Eclipse and IntelliJ IDEA.

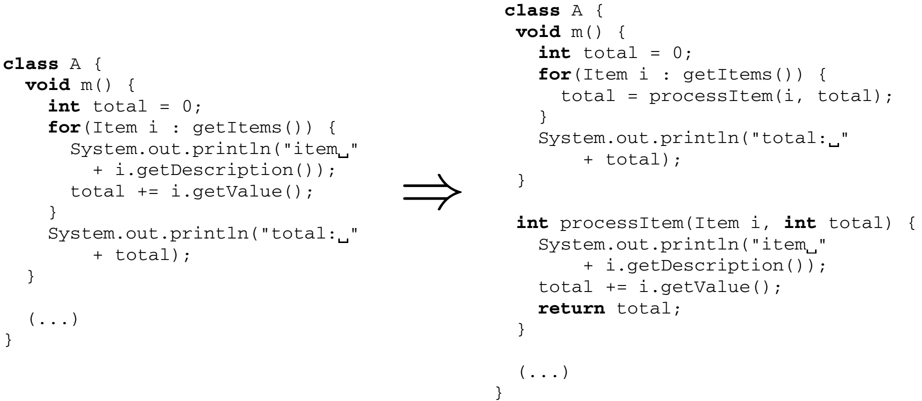
The main novel contributions of this paper are:

- A presentation of refactorings solely based on **invariant preservation**, which provides a more flexible implementation guideline than traditional precondition-based approaches.
- The use of **lightweight language extensions** as a device to simplify and modularise refactoring implementations.
- An **analysis of Extract Method** in terms of this general approach, showing how the overall refactoring can be decomposed into micro-refactorings that are easy to implement, understand, and test.
- A high-quality **implementation of Extract Method** and **related refactorings** based on this analysis, that is very compact, yet supports the whole Java 5 language and is on par with well-known Java IDEs in terms of correctness.

We structure this paper as follows: Section 2 introduces refactoring challenges, illustrated with the example of *Extract Method*. Section 3 shows how these challenges can be met using a decomposition of the whole refactoring into a series of well-defined micro-refactorings, and Section 4 provides a more in-depth discussion of how this can be done for method extraction. Section 5 puts these concepts into context and discusses how similar strategies can be used on related refactorings. Section 6 evaluates our implementation in terms of code size and correctness, comparing it against some other well-known Java refactoring engines. Section 7 discusses some related work directions for future work before we conclude in Section 8.

2 Challenges

We begin by introducing refactoring challenges through our running example, the *Extract Method* refactoring for the Java language. It is used to simplify complicated methods by extracting a contiguous range of statements into a new method, and replace the original statements by a call to that method. As a simple example, consider the method `m` in Figure 1 on the left, and assume we want to extract the body of the **`for`** loop into a new method `processItem`.



```
class A {
  void m() {
    int total = 0;
    for(Item i : getItems()) {
      System.out.println("item␣"
        + i.getDescription());
      total += i.getValue();
    }
    System.out.println("total:␣"
        + total);
  }

  (...)
}
```

```
class A {
  void m() {
    int total = 0;
    for(Item i : getItems()) {
      total = processItem(i, total);
    }
    System.out.println("total:␣"
        + total);
  }

  int processItem(Item i, int total) {
    System.out.println("item␣"
        + i.getDescription());
    total += i.getValue();
    return total;
  }

  (...)
}
```

Fig. 1. An example program before and after *Extract Method*

This is relatively easy; all we need to do is provide `i` and `total` as parameters to the new method, and return the value of `total` to update the original variable after the method returns. Thus the resulting program should look like the one shown in the same figure on the right.

But even this simple example shows that *Extract Method* is much more than just cut-and-paste. In general, we can identify three different kinds of problems to be handled, that can all be cast as preservation problems: we need to preserve name bindings, control flow, and data flow.

2.1 Name Binding Preservation

Since the refactoring introduces new names and declarations into the program, care has to be taken not to accidentally change existing name bindings. Take, for example, the program in Figure 2 on the left; the method `m` in class `A` constructs an instance of the locally declared class `X` and returns it.

In NetBeans, extracting the declaration of `X` to a method `n` yields the program on the right, where the instance constructed is no longer of the local class `X`, but of the global class of the same name. This particular program does not change its behaviour, but slightly extended examples either make NetBeans produce an output program that does not compile (which is annoying to the user, if comparatively harmless), or that still does compile but behaves differently.

```
class X { }
class A {
  Object m() {
    class X { }
    return new X();
  }
}
```

“⟹”

```
class X { }
class A {
  Object m() {
    return new X();
  }
  void n() {
    class X { }
  }
}
```

Fig. 2. How not to preserve name binding during method extraction

Other IDEs employ simple heuristics to guard against this kind of situation, but no systematic binding preservation seems to be attempted, and similar bugs can be discovered in all cases [3].

2.2 Control Flow Preservation

While Java mostly uses structured control flow, it also provides the unstructured branching statements **`break`** and **`continue`**. The former exits from an enclosing loop (which can be further specified by a label), while the latter only exits from the current iteration and starts the next one. These “**`goto`**s in disguise” cannot be moved into the newly created method blindly: If their target loop is not also moved, the resulting program will not compile.

Even more subtle is the **`return`** statement: If types match, an extracted method with an embedded return will still compile, but of course the statement now returns from the *extracted* method, not the original method as it did before. As an example, consider the program in Figure 3 on the left, and assume we want to extract the **`if`** statement into a method `n`. A naive implementation might produce the program on the right, which has different behaviour from the original program: while originally calling `m(23)` would return immediately, it now prints `23`.

A precondition-based approach might categorically forbid extraction of any code that contains these branching statements, but that would reject many potentially useful refactorings: As a simple example, consider the program in Figure 4 on the left, and assume we want to extract the whole body of `m` into a new method n. This can easily be done if, instead of replacing the extracted code by the method invocation `n(i)`, we instead replace it by **`return`** `n(i)`.

Thus a more advanced precondition might be to allow extraction if all control paths end in a **`return`** statement. But this is again too stringent a requirement, as the example

```
class A {
  void m(int i) {
    if(i == 23)
      return;
    System.out.println(i);
  }
}
```

“⟹”

```
class A {
  void m(int i) {
    n(i);
    System.out.println(i);
  }
  void n(int i) {
    if(i == 23)
      return;
  }
}
```

Fig. 3. How not to preserve control flow during method extraction

```
class A {
  int m(int i) {
    if(i == 23)
      return 42;
    return i + 1;
  }
}
```

(a)

```
class B {
  void x(int j) {
    if(j == 42)
      return;
    System.out.println(j);
  }
}
```

(b)

Fig. 4. Preserving control flow during method extraction

program in the same figure on the right shows. Our preconditions would not allow us to extract the whole body of `x` into a new method `y`, although this would actually be unproblematic, since the code to be extracted is right at the end of the enclosing method anyway.

These examples show that precondition based refactoring engines are doomed to play an ultimately pointless game of tag in which preconditions have to be progressively relaxed as desirable refactorings are discovered. It is all too easy to introduce unsoundness this way, especially if new language versions introduce constructs that influence control flow (such as the **`assert`** statement in Java 1.4).

We instead propose preservation of control flow as the goal to aim for during method extraction: every statement in the extracted method should have the same control flow predecessors and successors as before the extraction. This immediately rules out **`break`** and **`continue`** statements whose target is not in the new method: their control flow successors will no longer be defined at all. For the two examples above, it can however be seen that their control flow successors do not change during the proposed refactorings, although flow sometimes has to be "rerouted" by inserting extra statements like the **`return`** of the first example.

Since refactorings work on a source-level representation of the program, not its generated byte code, some care has to be taken in determining control flow information, which we will discuss in more detail in Section 3.

2.3 Data Flow Preservation

One of the most intricate aspects of the *Extract Method* refactoring is to determine the parameters of the extracted method. Intuitively, it is clear that we need to pass the new method the values of any local variables it might need, and that the new method in turn should return the values of any local variables it has changed, if they are needed for further computation.

In previous work [23] we have given criteria for selecting parameters and return values in terms of familiar data flow concepts such as liveness and def/use sites. As it turns out, these criteria can be given a more basic justification in terms of data flow preservation.

Analogous to control flow successors and predecessors, we define data flow successors and predecessors: The data flow predecessors of a variable use are its reaching

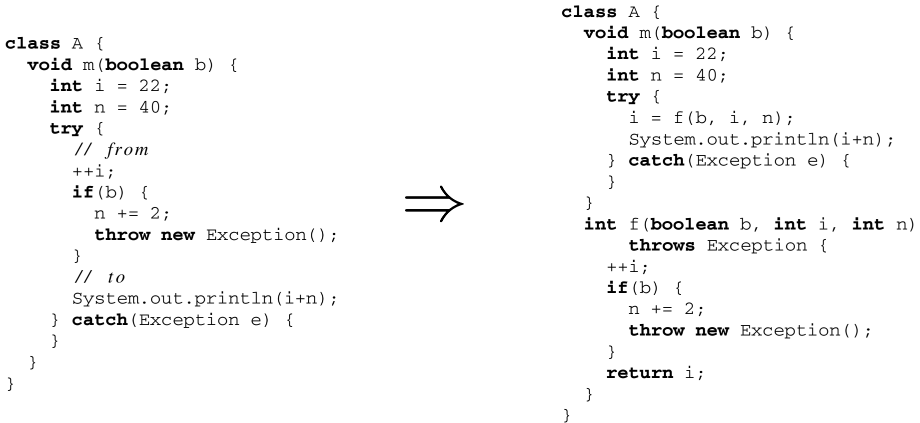


Fig. 5. A refactoring rejected by Eclipse

definitions, i.e. all definitions of this variable that might influence the value of the variable at this point in the program. Similarly, the data flow successors of a variable definition are all uses that might be reached without an intervening definition.

We can now specify that a variable should become a parameter to the extracted method if it has a use within the code to be extracted whose data flow predecessor lies before the extracted selection, and it should be returned if it has a definition whose data flow successor comes after the selection. This is a natural criterion, since those data flow edges would be "broken" by method extraction, and hence have to be rerouted through parameters in order to be preserved.

Again, we leave it to Section 3 to make this basic idea more precise and show how it can be implemented at the abstract syntax tree level. That this is not a trivial problem can be illustrated by a simple example: Assume that in the program in Figure 5 on the left we want to extract the code between the comments. We note that all of `b`, `n`, and `i` have to become parameters to the new method, and `i` should be returned so that its new value is available for the `println` statement, as shown in the program on the right.

The value of `n`, however, does *not* need to be returned: if `n` is changed by the extracted code, an exception is immediately thrown and control transfers to the **`catch`** clause; in other words, the assignment to `n` has no data flow successor, and hence its value does not need to be returned. Eclipse, for example, does not detect this, and determines that both values need to be returned. This is not easily accomplished in Java, and hence the refactoring is rejected.

3 Our Approach

As outlined in the last section, the main challenge in implementing the *Extract Method* refactoring and its brethren is the intertwining of name binding, control flow, and data flow, which are all delicate by themselves, and even more so in conjunction.

By using our existing naming framework, we can easily achieve binding preservation where necessary, so we can concentrate on preserving control and data flow. It is a

natural idea to simplify this task by splitting the refactoring into two parts that deal with control flow and data flow separately. Unfortunately, this is difficult to achieve in plain Java: At some point during the refactoring, the statements to be extracted have to be moved into a new method, and at that point both their control and data flow will change at the same time.

We hence introduce *anonymous methods* into the language as a lightweight extension that helps breaking up the transformation. Just as an anonymous class is a class that is defined and instantiated at the same time and may access local variables from the surrounding method, an anonymous method is a method that is defined and invoked at the same time, and (besides its own parameters and local variables) can also access variables from the surrounding method.

In control flow terms it behaves like an ordinary method, in particular **break** and **continue** statements can not escape the anonymous method. In data flow terms, however, it behaves like a block in that it can access variables from the enclosing scope in a lexically scoped fashion.

Thus an anonymous method provides a convenient half-way point for the *Extract Method* refactoring: If we can package up the statements to be extracted into an anonymous method, it means that we have successfully preserved the control flow. We can then tackle the task of preserving data flow by successively introducing parameters and return values as needed. Once the anonymous method does not reference any local variables from the surrounding method anymore, it can safely be promoted to a normal method without disturbing either control or data flow.

While we leave a detailed exposition of the different steps involved in extracting a method to the next section, we will briefly describe the general procedure by using our initial example. Here is the original program, with the statements to be extracted framed by comments:

```
class A {
  void m() {
    int total = 0;
    for(Item i : getItems()) {
      // from
      System.out.println("item␣" + i.getDescription());
      total += i.getValue();
      // to
    }
    System.out.println("total:␣" + total);
  }

  (...)
}
```

As a first step, it will be convenient to turn the sequence of statements to extract into a block so that we can analyse it and operate on it as a single abstract syntax tree. This first step, a micro-refactoring we will refer to as *Extract Block*, is fairly easy to implement, since blocks do not affect control and data flow, and yields the following program:

```
class A {
  void m() {
    int total = 0;
    for(Item i : getItems()) {
    {
      System.out.println("item␣" + i.getDescription());
```

```
      total += i.getValue();
    }
    }
    System.out.println("total:␣" + total);
  }

  (...)
}
```

Now we turn the extracted block into an anonymous method without parameters, carefully preserving control flow in the process. This micro-refactoring, called *Introduce Anonymous Method*, yields this:

```
class A {
  void m() {
    int total = 0;
    for(Item i : getItems()) {
      (() : void ⇒ {
        System.out.println("item␣" + i.getDescription());
        total += i.getValue();
      })();
    }
    System.out.println("total:␣" + total);
  }

  (...)
}
```

We write the anonymous method as `(() : void ⇒... )()`, indicating that it has no formal parameters, is of return type **void**, and is applied to an empty list of arguments. Note that this syntax is for presentation purposes only, and anonymous methods cannot appear in user programs.

Our next task is to reroute dataflow edges that go across the boundaries of the anonymous method, i.e. to do a lambda lifting [11]. To ease this step, we allow our anonymous methods to have *reference* parameters like in C#. Thus we get the following:

```
class A {
  void m() {
    int total = 0;
    for(Item i : getItems()) {
      ((int i, ref int total) : void ⇒ {
        System.out.println("item␣" + i.getDescription());
        total += i.getValue();
      })(i, total);
    }
    System.out.println("total:␣" + total);
  }

  (...)
}
```

The anonymous method now has two parameters `i` and `total`, both of type **int**, to which the arguments `i` and `total` from the surrounding method are assigned. In the former case, the parameter `i` and the argument `i` are entirely different entities, whereas in the latter case the reference parameter `total` is aliased to the variable `total` to make sure that any changes to the parameter are reflected in the variable.

Before we can lift the anonymous method to a named method, we need to eliminate reference parameters, which are not supported in normal Java. The easiest way is to require that there be at most one such parameter, and turn it into a return value, yielding the following result:

```
class A {
  void m() {
    int total = 0;
    for(Item i : getItems()) {
      total = ((int i, int total) : int ⇒ {
        System.out.println("item␣" + i.getDescription());
        total += i.getValue();
        return total;
      })(i, total);
    }
    System.out.println("total:␣" + total);
  }

  (...)
}
```

Now the anonymous method's body does not reference any local variables from the surrounding method, thus its control and data flow both behave as with a named method, and we can, in a final step, lift it out to complete the extraction.

```
class A {
  void m() {
    int total = 0;
    for(Item i : getItems()) {
      total = n(i, total);
    }
    System.out.println("total:␣" + total);
  }
  int n(int i, int total) {
    System.out.println("item␣" + i.getDescription());
    total += i.getValue();
    return total;
  }

  (...)
}
```

We would like to particularly emphasise that we do not propose to add anonymous methods as a general mechanism to the Java language standard. It is rather a lightweight extension that we use to simplify the implementation of refactorings. There is no parsing syntax for anonymous methods, hence they can never occur in user programs, and no code can be generated for them. All we need to specify is their behaviour in terms of name binding, control flow, and data flow.

Although we have described only the *Extract Method* refactoring here and will continue to concentrate on it as our running example, other refactorings that deals with code movement, in particular *Inline Method*, *Extract Local Variable*, and *Inline Local Variable*, face exactly the same kind of problems, and can indeed be treated in the same way.

4 *Extract Method* in Small Pieces

Using the concepts introduced in the previous section, we will now take a more detailed look at the implementation of *Extract Method*. Our refactoring engine is implemented as an extension to the JastAddJ Java compiler [4], hence it can work on the abstract syntax tree produced by the compiler frontend, and use all of the static analysis machinery provided by the compiler. In particular, this includes name lookup to easily navigate from a name to its declaration.

We also make use of a new implementation of the intra-procedural control flow analysis library presented in [18]. This library provides for every AST node two attributes `pred` and `succ`, that compute the nodes preceding resp. succeeding this node in terms of control flow. In particular, this allows us to compute, for any statement, its preceding and succeeding statements. For a given node `n`, if the node `s` is contained within the set `n.succ()`, we say that there is a "*succ* edge" from `n` to `s`, and similar for `pred`.

We can now implement a "locking" and "unlocking" mechanism somewhat similar to the bound names framework [21]: Before performing a transformation, we compute the control flow predecessors and successors of the statements that the transformation acts upon. Then after the transformation we recompute this information and verify that it has not changed, aborting the refactoring if any change has occurred[1].

A certain category of syntax tree nodes can be categorised as *flow-through* nodes, which purely contribute to the program's control flow without influencing any other aspect of the semantics. Examples are nodes corresponding to **`break`** and **`continue`** statements, but also nodes corresponding to **`return`**[2]. When checking for control flow preservation, we disregard such nodes, thus creating the opportunity to "fix" control flow by inserting additional flow-through nodes.

Note that the *post hoc* flow preservation check makes the refactoring robust in the face of future language extensions: if additional control flow constructs appear which are not handled by the refactoring, the transformation will be aborted instead of producing incorrect results.

On top of the control flow analysis we have implemented a lightweight dataflow analysis for local variables and parameters which provides two attributes `dataPred` and `dataSucc` for every variable node in the syntax tree. The former computes all assignment statements for the same variable that can be reached by walking along `pred` edges without encountering any intervening assignments to this variable; the latter does the reverse and computes a set of variable nodes that may use the value assigned to a variable in an assignment. Thus, `dataPred` gives us all reaching definitions of a variable use, whereas `dataSucc` computes the set of reached uses of a variable definition.

This flow analysis framework, together with the naming toolkit, provides the basis on which we build our implementation of method extraction. The extraction process is split up into five smaller refactorings:

1. *Extract Block* pulls the statements to be extracted together into a block.
2. *Introduce Anonymous Method* turns that block into an anonymous method without parameters.
3. *Close Over Variables* eliminates any references to local variables from within the anonymous method by introducing parameters.
4. *Eliminate Reference Parameters* gets rid of parameters that need to be passed by reference, since this is not supported by Java.

[1] It is enough to perform intra-procedural flow analysis, since the naming framework guarantees that method calls are resolved to the same method before and after the refactoring, which means that inter-procedural control flow will not change.

[2] However, for a statement **`return`** `e;`, the node corresponding to `e` (and its children) are *not* flow-through, since they correspond to actual computation taking place.

5. *Lift Anonymous Method* turns the anonymous method into a named method within the same type as the method we are extracting from.

The whole *Extract Method* refactoring is simply a sequential composition of these five sub-refactorings, which we now are going to discuss in greater detail. The control flow and data flow analyses are run before and after each refactoring step to ensure that the invariants are preserved.

4.1 Extract Block

The *Extract Block* refactoring takes as its input a block b (represented by its node in the abstract syntax tree) composed of statements b_1 to b_n and two indices i and j such that $1 \leq i \leq j \leq n$. The goal of the refactoring is to put statements b_i to b_j into a new block b' and insert it into b to replace the original statements.

This refactoring can itself be composed from even simpler operations: In a first step, we insert an empty block after statement b_j, then we successively move the statements b_j to b_i into this new block. For every statement to be moved, we want to ensure that its flow and bindings do not change, so we lock flow and binding information, move it into the block, and then unlock it; this operation can be encapsulated into a micro-refactoring *Push Statement into Block*.

As an example, consider the program in Figure 6 (a), and assume we want to wrap the first three statements of m into a block. We create an empty block and push the third

```
class A {
  int x = 23;
  int m() {
    x = 42;
    int x = 55;
    ++x;
    return x;
  }
}
```

(a)

```
class A {
  int x = 23;
  int m() {
    x = 42;
    int x = 55;
    {
      ++x;
    }
    return x;
  }
}
```

(b)

```
class A {
  int x = 23;
  int m() {
    x = 42;
    int x;
    x = 55;
    {
      ++x;
    }
    return x;
  }
}
```

(c)

```
class A {
  int x = 23;
  int m() {
    int x;
    this.x = 42;
    {
      x = 55;
      ++x;
    }
    return x;
  }
}
```

(d)

```
class A {
  int x = 23;
  int m() {
    int x;
    {
      this.x = 42;
      x = 55;
      ++x;
    }
    return x;
  }
}
```

(e)

Fig. 6. *Extract Block* in action

statement into it, which does not violate binding or flow preservation and yields the intermediate program in (b).

The next statement cannot be pushed into the block directly: it declares the variable `x` which is referenced after the block; if we were to push the whole statement into the block, `x` would become invisible, and the resulting program would fail to compile. Hence we first employ a micro-refactoring *Split Variable Declaration*, that turns a variable declaration with initialisation into a "pure" declaration without initialiser followed by an assignment, yielding program (c).

Now the statement `x = 55;` can be pushed into the block without any problems. We also need to move the declaration up to the beginning of the selection to clear the way for pushing the remaining statements. This, of course, creates a problem, since the local variable `x` would now shadow the field `x` in the assignment `x = 42`. Fortunately, our naming framework takes care of this automatically, and after unlocking we get (d).

Moving the last statement into the block does not present any particular difficulties, and we obtain our final program in (e).

Of the three minor refactorings discussed here, *Extract Block* might be considered a useful refactoring in its own right to clarify code structure. *Push Statement Into Block* and *Split Declaration*, on the other hand, are pure building blocks that are not very useful to the programmer; they can, however, be reused in the context of other refactorings.

4.2 Introduce Anonymous Method

Next we want to convert the block created in the previous step into an anonymous method without parameters. JastAddJ, of course, has no built-in support for anonymous methods, since they are not part of the Java language. Its very extensible implementation, however, makes it extremely easy to add language extensions.

Support for anonymous methods can be added by providing an additional production for an abstract syntax tree node in the object language:

```
AnonymousMethod : Expr ::= Parameter:ParameterDeclaration*
      ReturnType:Access Exception:Access* Block Arg:Expr*;
```

In words, this production says that anonymous methods are a kind of expression, i.e. they can occur anywhere the Java language grammar allows an expression to occur. They contain a list of parameters, represented by the same node type as parameters for plain Java methods and constructors; a return type which is an `Access`, i.e. a possibly qualified name; a list of thrown exceptions, likewise given as accesses; a body given as a `Block`; and finally a list of arguments, which may be arbitrary expressions.

We do not specify any parsing rules for anonymous methods, since we do not want to make them available for programmers, but for presentation purposes we use the concrete syntax $(\overline{p} : r \textbf{ throws } \overline{x} \Rightarrow b)(\overline{e})$ to represent an anonymous method with body b that takes parameters $\overline{p}$, is invoked with arguments $\overline{e}$, throws exceptions $\overline{x}$ and has return type r.

To defer the handling of multiple return values, we introduce another language extension in the form of *output* and *reference* parameters, marked with the modifiers **`out`** and **`ref`**, respectively. The argument given for such parameters must be a variable of the

enclosing scope, and any changes the anonymous method makes to the parameter are reflected in that argument, thus these arguments are (conceptually) passed by reference.

Parameters marked **`ref`** may be read before they are assigned, which is not possible for **`out`** parameters. These two kinds of parameters behave like their counterparts in C#, but they only occur as ephemeral language constructs during refactoring, not as genuine language features. A parameter that is marked neither **`out`** nor **`ref`** is called a *value parameter*.

Conceptually, control flow for an anonymous method works like for a normal method call: parameters are bound to arguments, the body is executed, and may return a value by executing a **`return`** statement. Exception handling likewise works as for methods, exceptions thrown but not caught within the anonymous method propagate to the enclosing scope and onwards until a corresponding **`catch`** clause is found. Like a normal method, an anonymous method may declare and use local variables in addition to its parameters, and it may also access any variable or field visible in the surrounding method, subject to lexical scoping.

To turn a block into an anonymous method without parameters, we do not need to adjust any data flow or name bindings: all these work the same way for blocks as they do for anonymous methods. We do, however, need to make sure that control flow is preserved. Hence, we lock down control flow in the block, computing and caching the predecessor and successor statements of every statement in the block, then wrap it into an anonymous method, and unlock control flow, recomputing all predecessor and successor statements to make sure they have not changed.

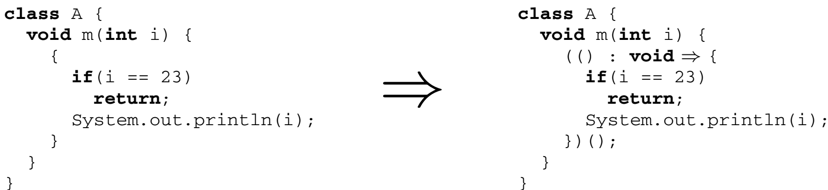
More precisely, given a block b in a context with return type T[3], we perform the following steps:

1. Lock all control flow in b.
2. Compute all uncaught checked exceptions thrown in b, and use the naming framework to compute locked accesses $e_1, \ldots, e_n$ for them.
3. Construct an anonymous method C of the form

   `(() :` R **`throws`** $e_1, \ldots, e_n \Rightarrow$ `b)()`

   where R is T if b cannot complete normally[4], or **`void`** otherwise.
4. If b can complete normally, replace it by C`;`, otherwise by **`return`** C`;`.
5. Unlock control flow, aborting the refactoring if the flow has changed.

In the second example program of Figure 4, for example, the block can complete normally, so we perform the following transformation:

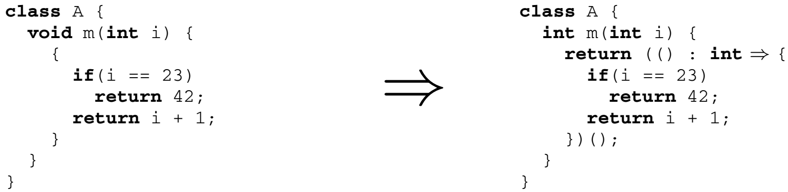


[3] That is, T is either the return type of the enclosing method, or it is **`void`** if b is not inside a method.

[4] That is, if every control flow path through b ends in a control transfer statement like **`return`**; for the precise definition see the Java Language Specification[8].

In the program on the left, the only control flow successor of both the **return** statement and the `println` statement is the “exit node” of the enclosing method, which is a pseudo-statement indicating that control flow has reached the end of the method (see [18] for more details on this point). The same is true in the refactored program on the right, and since all the other predecessor and successor statements are likewise preserved, the refactoring can continue.

In the first example program of Figure 4, the block *cannot* complete normally (both control paths end in a **return** statement), hence we transform as follows:



```
class A {                              class A {
  void m(int i) {                        int m(int i) {
    {                                      return (() : int ⇒ {
      if(i == 23)             ⟹              if(i == 23)
        return 42;                               return 42;
      return i + 1;                            return i + 1;
    }                                      })();
  }                                      }
}                                      }
```

Again, we can verify that control flow successors have not changed: the newly inserted **return** is a flow-through node, and hence does not disrupt control flow.

4.3 Close over Variables

We now want to perform lambda lifting on the anonymous method produced by the previous step, to make explicit any dependencies on local variables from the surrounding scope. In data flow terms, we need to handle two situations:

– The anonymous method might read a local variable `x` from the surrounding scope, whose value was set before executing the method; thus it has an incoming dataflow edge that crosses the boundaries of the anonymous method.
– The anonymous method might write a local variable `y` from the surrounding scope whose value is read after the method has finished executing; thus it has an outgoing data flow edge that crosses the method’s boundaries.

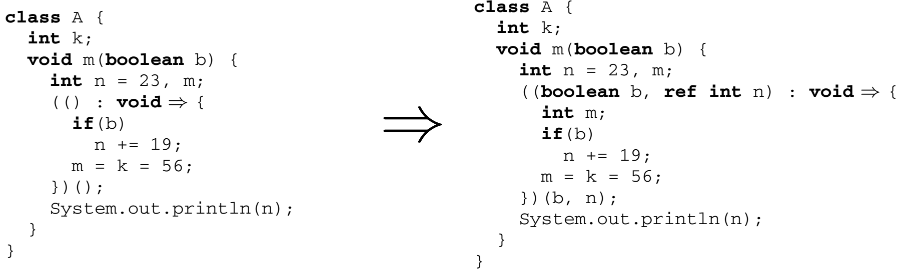
In the first case, `x` needs to be made a value parameter of the method, in the second case `y` should become an output parameter. Of course, both situations may apply to the same variable, which should then be classified as a reference parameter.

We can implement these conditions in terms of the dataflow framework: For a given occurrence of a variable `x` in the anonymous method,

– `x` should be made a value parameter if any of its data flow predecessors come before the entry of the anonymous method,
– `x` should be made an output parameter if any of its data flow successors come after the exit of the anonymous method,

and it should be made a reference parameter if both situations apply.

After all variables have been treated in this manner, we are guaranteed that all data flow edges have been safely rerouted. There might, however, still be references to local variables from the enclosing scope in the body; this happens if the use of a local variable



```
class A {
  int k;
  void m(boolean b) {
    int n = 23, m;
    (() : void ⇒ {
      if(b)
        n += 19;
      m = k = 56;
    })();
    System.out.println(n);
  }
}
```

⇒

```
class A {
  int k;
  void m(boolean b) {
    int n = 23, m;
    ((boolean b, ref int n) : void ⇒ {
      int m;
      if(b)
        n += 19;
      m = k = 56;
    })(b, n);
    System.out.println(n);
  }
}
```

Fig. 7. Closing over local variables

inside the anonymous method is independent of its use in the enclosing method, and we can then safely make it a local variable of the anonymous method instead.

As an example, consider the program in Figure 7 on the left. The *Close Over Variables* refactoring considers the three references to `b`, `n`, and `m` inside the anonymous method (but not the reference to `k`, since it is a field).

- `b` has an incoming dataflow edge; the parameter declaration counts as a definition, there are no intervening definitions of `b`, and the definition is outside the anonymous method. It has no outgoing dataflow edge, since it is never used afterwards.
- `n` has both an incoming and an outgoing dataflow edge; its data flow predecessor is its declaration, which is outside the anonymous method; its data flow successor is the `println` statement, which is likewise outside the anonymous method.
- `m` has neither incoming nor outgoing dataflow edges, since it has no data flow predecessors or successors.

Thus, `b` should be made a value parameter, `n` a reference parameter, and `m` a local variable, yielding the program on the right.

Note that control flow cannot be influenced by the transformations done in this step, so no control flow locking and unlocking is needed. Also, we forgo binding preservation in this step: uses of local variables from the surrounding scope will now bind to the corresponding parameters instead. In all the preceding (and most of the following) steps, however, we do indeed want to preserve binding. Our decomposition of the refactoring helps to clarify this situation and makes it possible to precisely pinpoint at which stages binding preservation is required, and where we have deliberately chosen to relax it.

4.4 Eliminate Reference Parameters

Our refactoring process has now reached its apogee from the Java language specification: not only are we working with an anonymous method, a construct unknown to the JLS, but it even might feature output and reference parameters, which are likewise not supported by the language. It is now our task to safely remove these extra features to bring our program back into the fold of standard compliant Java programs.

In this step, we eliminate **out** and **ref** parameters. The basic idea is that a **ref** parameter can be simulated by a value parameter whose value is returned to the surrounding method, and there assigned to the corresponding variable; an **out** parameter is a local variable whose value is passed back in the same fashion.

More precisely, we perform the following steps:

1. If there is no output or reference parameter, the refactoring is a no-op. If there is more than one such parameter, abort (since Java methods can only return a single value).
2. If the anonymous method already has a non-**void** return type, abort likewise, for the same reason.
3. Let `x` be the only non-value parameter. If it is a reference parameter, change it into a value parameter. Otherwise make it into a local variable of the anonymous method and remove the corresponding argument.
4. Change the return type of the anonymous method to the type of `x`.
5. Insert a statement to return `x` and wrap the whole anonymous method into an assignment to the corresponding variable in the surrounding method.

All these steps are quite straightforward to implement; for example, the program from Figure 7 becomes

```
class A {
  int k;
  void m(boolean b) {
    int n = 23, m;
    n = ((boolean b, int n) : int ⇒ {
      int m;
      if(b)
        n += 19;
      m = k = 56;
      return n;
    })(b, n);
    System.out.println(n);
  }
}
```

As explained, this refactoring step rejects any anonymous method that needs to return the value of two or more variables, which results in a behaviour similar to Eclipse. An alternative would be to package up the necessary return values into a wrapper object, return that object, and unwrap it again in the calling method.

4.5 Lift Anonymous Method

To finally turn our refactored program back into a normal Java program, we need to eliminate the anonymous method. We know that it has only value parameters and its body does not reference any local variables from the enclosing scope; hence it is semantically equivalent to a call to a named method with the same body and parameters and the same arguments.

Assuming that we want to extract the anonymous method to a method named `f`, we simply convert $(\overline{p} : r \textbf{ throws } \overline{x} \Rightarrow \overline{b})(\overline{e})$ into the method call `f`$(\overline{e})$, and insert the definition

r `f(`$\overline{p}$`)` **throws** $\overline{x}$ `{`
  $\overline{b}$
`}`

into the surrounding class declaration. Our naming framework is put to use to ensure that the call to `f` really binds to the newly inserted method `f` (and that there are no overriding methods), and that all type bindings are preserved; this will, for example, detect (and reject) a refactoring that tries to extract a statement that throws a locally declared exception type.

The example program from above, of course, poses no such problems, and we can successfully complete the refactoring:

```
class A {
  int k;
  void m(boolean b) {
    int n = 23, m;
    n = f(b, n);
    System.out.println(n);
  }
  int f(boolean b, int n) {
    int m;
    if(b)
      n += 19;
    m = k = 56;
    return n;
  }
}
```

4.6 Putting It All Together

The micro-refactorings introduced above are all implemented as methods on AST node types, which are just Java classes. For example, the *Extract Block* refactoring is a method in class `Block` with the signature

```
Block extractBlock(int i, int j) throws RefactoringException
```

When invoked as `b.extractBlock(i, j)`, the method extracts statements `i` to `j` of block `b` into a new block, and returns it as a result. If, at any point, the refactoring cannot proceed (for example due to an invariant violation), an exception is thrown.

The other refactorings are implemented in a similar way, so that the complete *Extract Method* refactoring can be realised as a one-liner (although we introduce some line breaks for æsthetic reasons):

```
MethodDecl extractMethod(int i, int j, String n, String v)
    throws RefactoringException {
  return extractBlock(i, j).extractAnonymous().
         closeVariables().eliminateOut().lift(n, v);
}
```

This method again belongs to class `Block`, and is invoked on the block from which we want to extract statements `i` to `j`. The parameter `n` determines the name of the new method, and `v` its visibility. Since all the micro-refactorings return the result they produce, an invocation chain can be used to implement their sequential composition.

Space constraints prevent us from showing more code, but the entire refactoring framework is available at `http://jastadd.org/refactoring-tools`.

5 Discussion

The previous two sections have discussed in some detail an implementation of *Extract Method* as a sequential composition of five component refactorings that make essential use of several lightweight language extensions to achieve better separation of concerns, and to get a clearer picture of how the refactoring preserves certain syntactic invariants to achieve behaviour preservation.

The same approach works equally well for other refactorings. Let us take a short look at *Inline Method*. It is the inverse of *Extract Method* in that it takes an invocation of a method and inlines the method body, substituting it for the invocation expression. It can likewise be decomposed into five micro-refactorings, that are precisely inverse to their counterparts from *Extract Method*:

1. *Anonymise Method* turns a method call that can be statically resolved into an anonymous method with the same body, parameter list and exception list as the named method being invoked, applied to the same arguments.
2. *Introduce Reference Parameter* removes return statements and replaces them with assignments to an output parameter.
3. *Open Variables* substitutes actual arguments for formal parameters, using the locking framework to ensure that name binding and data flow remain unaffected.
4. *Inline Anonymous Method* replaces an invocation of an anonymous method without parameters by its body, preserving control flow.
5. *Inline Block* inlines all statements within a block into the surrounding block.

As another example of a similar pair of inverse refactorings we have *Extract Local Variable*, which extracts an expression into a local variable, and *Inline Local Variable*, which eliminates all uses of a local variable by substituting its value. Both were very easy to implement in our framework.

In all these cases, the use of anonymous methods allows us to separate the concerns of data flow preservation and control flow preservation, and as before both the name binding framework and the control and data flow library see heavy use. Several smaller language extensions are also used: For example, when inlining method parameters during *Open Variables* it is useful to introduce temporary local variables whose names should not conflict with any other variables in scope. We introduce such non-shadowing variables as an additional language construct which can be turned into a normal declaration by choosing a fresh name.

These language extensions only exist at the level of abstract syntax. It is hard to imagine a good syntactic representation, e.g., for a bound name that explicitly specifies its binding target, and it would certainly not be conducive to a source program's clarity to explicitly contain such constructs. But that is not their goal: Bound names and bound control flow exist only to raise the level of abstraction for implementing refactorings and to address in one place issues that occur in several different refactorings.

The situation is a bit different for anonymous methods and their support for reference and output parameters: again, these only exist in the abstract syntax tree and cannot be used in source programs. Programming languages like C# or Smalltalk, on the other hand, come with built-in support for this kind of language feature, which could potentially simplify the implementation of refactorings like *Extract Method*. However, it

may be the case that the complexity of handling closures in the analysis outweighs the benefit of using them for decomposition.

It is perhaps interesting to observe that our use of an intermediate language for refactoring goes against the usual trend: For static analysis, for example, one would normally choose an intermediate language that is *simpler* than the source language, abstracting away from its idiosyncrasies and simplifying it down to a core language. We work instead on a *richer* language that introduces new features which we find useful for structuring the refactoring process.

However, this does not complicate the implementations of the refactorings proper. While there are numerous statements in Java that affect control flow, from the refactoring point of view it is sufficient to abstract them as nodes in a control flow graph. Similarly, all expressions in Java that contain an assignment as a side effect can be abstracted as reaching definitions.

The challenge of refactoring a rich language with various extensions thus boils down to implementing name binding, control flow analysis, and data flow analysis for the full language. These analyses can all be implemented in a modular and extensible way, taking advantage of the compiler infrastructure [21,18].

While the analyses provide these abstractions and use them to enforce invariants, the micro-refactorings build on top of these abstractions and therefore only need to deal with a much smaller language. For example, *Extract Method* needs to be aware of concepts such as methods, parameters, return values, and exceptions; the remaining language features can be abstracted as their effect on control and data flow.

This abstract view is sufficient for the refactoring developer. The invariant checking provides a safety net that guards against corner cases and surprising interactions between language features. For example, if the developer did not implement the *Close Over Variables* micro-refactoring, method extraction would still be possible for a limited number of cases, and reject cases were lambda lifting is needed.

The developer could then further improve the transformation to cover more cases without the danger of inadvertently allowing unsound refactorings. This stands in sharp contrast to a precondition-based approach where unanticipated situations easily lead to incorrect transformations.

6 Evaluation

For evaluating our approach, we first present some statistics on the amount of code needed to implement the discussed refactorings.

The naming framework is the biggest component, requiring about 1700 lines of code[5]; this excludes code that is specific to the *Rename* refactorings. The implementation of the control flow analysis contributes around 550 lines of code, whereas the data flow analysis for local variables is implemented in 200 lines of code. Note that the former two components were written well before we started implementing the refactorings discussed in this paper, only the data flow analysis was implemented specifically for this project.

[5] This and all following code size measurements were generated using David A. Wheeler's 'SLOCCount'.

The implementation of anonymous methods requires about 150 lines of code, for the definition of corresponding AST node types, pretty printing, name analysis, and control and data flow analysis.

The size of the individual refactorings we have implemented is summarised in Figure 8: for each refactoring we show its decomposition in micro-refactorings, and the size of each micro-refactoring.

Extract Method takes around 500 lines of code, quite evenly distributed between the individual micro-refactorings: *Extract Block* needs about 140 lines (including the code for *Push Statement Into Block* and *Split Declaration*), *Introduce Anonymous Method* and *Close Over Variables* around 100 lines each, whereas *Eliminate Reference Parameters* and *Lift Anonymous Method* each are implemented in less than 90 lines.

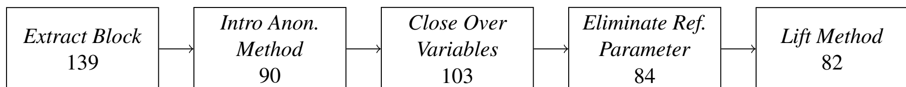
Similar numbers obtain for *Inline Method*, shown in the same figure on the right, which is overall a bit smaller than *Extract Method*; *Extract Local Variable* and *Inline Local Variable* are significantly smaller, at about 80 lines of code each.

These numbers compare very favourably both to Eclipse's implementation of *Extract Method*, which comprises more than 1500 lines of code, and other implementations like the one presented by Juillerat [12] that seems to offer less functionality for only a subset of Java at around 1000 lines of code.

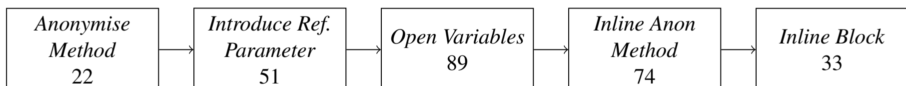
To test the correctness of our refactorings, we put together a suite of about 90 tests for *Extract Method* and its constituent micro-refactorings, and several dozen for the other refactorings, in particular including all the test cases for bugs in the refactoring engines of popular IDEs [3].

We also ran our engine on the test suite for Eclipse 3.4, which is publicly available, and some tests from the test suite for IntelliJ IDEA 8.0, which JetBrains kindly provided for us to use.

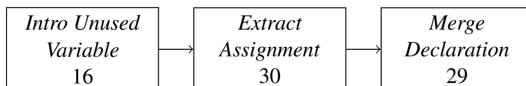
Extract Method:



Inline Method:



Extract Local Variable:



Inline Local Variable:

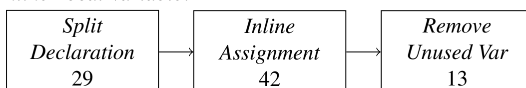


Fig. 8. Structure and code size for four refactorings

The former includes 395 test cases for *Extract Method*, of which 23 test functionality that we do not support yet: Two test cases concern the extraction of a method into a surrounding class other than the immediate host class. Another 21 cases test duplicate elimination, where a number of occurrences of the same expression are jointly extracted into a method. This feature is largely orthogonal to the extraction process proper and relies on clone detection [1], which we have not implemented in our framework yet. All of the remaining 372 test cases are handled correctly by our implementation, among them three on which Eclipse's own refactoring engine fails with a null pointer exception, and one on which it produces a wrong result.

The test cases that JetBrains provided to us comprise 74 tests for *Extract Method*, of which 19 again concern duplicate elimination. We pass all the remaining 55 tests.

Where our engine did not produce identical code to Eclipse or IntelliJ, we manually checked to make sure that our output is equally valid. Such discrepancies are mostly due to a different ordering of the parameters of the extracted method or similar syntactic variations.

In summary, we can say with confidence that our implementation is at least as good as these two industrial-strength implementations, and in some cases manages to detect and properly handle situations they fail to address.

7 Related and Future Work

Refactoring literature usually presents statement-level refactoring like *Extract Method* as primitive transformations useful in the composition of larger-scale refactorings [19,13]. We are, however, by no means the first to advocate the decomposition of that kind of refactoring into yet smaller components. In [20], Perera paints a compelling picture of how the use of micro-refactorings changes the way refactorings are used by the programmer, with method extraction in Java as one of his examples. His proposed building blocks, *e.g. Push Statement into Method* for moving statements one by one into a freshly created method, are still quite complex, however, as they still need to yield a valid Java program at each intermediate step. It is hence not obvious that his decomposition actually makes it easier to implement a behaviour preserving refactoring engine. By comparison, our use of lightweight language extensions makes the individual micro-refactorings much simpler and easier to think about and implement.

Such decomposition of statement-level refactorings is also addressed by Ettinger in his inspirational PhD thesis [5], in which he develops a theoretical framework for slicing-based behaviour-preserving transformations. Noteworthily, he chose to embed the concept of liveness directly into programs, by explicitly stating variables that cannot be local to a statement. Rather than tackling a full mainstream language like we do, however, Ettinger focuses on proving the correctness of transformations for programs written in a simple *ad hoc* imperative language.

The concept of control and data flow preservation features prominently in the influential paper of Komondoor and Horwitz [14] on procedure extraction. Under their idealising assumptions, they can indeed prove that flow preservation entails semantics preservation, which, as we have discussed, is not the case for Java. However, their work mostly focuses on moving statements together to arrange them into a contiguous block

suitable for extraction; the actual process of extracting into a procedure they dismiss as "straightforward".

Our use of language extensions like bound names bears a certain resemblance to work by Mens and others on implementing transformations on graph representations of programs [16]. The graph they propose, however, only encodes part of the program, namely its high-level structure, as none of the refactorings they discuss actually require control and data flow information. Furthermore, their work does not address the problem of translating back the graph representation into a source-level program, which is essential for the purposes of a refactoring engine.

The composition of smaller refactorings into larger ones has been the focus of work by Kniesel and Koch [13]. For the refactorings we have considered in this paper, a very simple form of sequential composition is sufficient, which they call AND-sequence. Their work is also concerned with composing the pre- and post-conditions of constituent refactorings, which does not directly apply to our invariant-based presentation.

An invariant-based approach to structural refactoring has been suggested some time ago by Griswold [9]. He provides a small catalogue of simple "program restructurings" that preserve data and control dependencies as captured in a Program Dependence Graph [15]. While these restructurings are quite similar in scope and intent to our micro-refactorings, they are presented for a very small and well-behaved object language (a first-order subset of Scheme), where they are much easier to implement and reason about than with Java.

For future work, we would like both to explore the implementation of new refactorings and to extend our current implementations. There is still a number of structural refactorings that our engine does not support, for example *Encapsulate Field* or *Move Method*, but it seems that most of these should be easy to implement using our by now well-developed toolbox of name binding and flow analyses.

A more interesting challenge would be to implement the *Extract Slice* refactoring [6], which does away with the restriction of *Extract Method* that only consecutive statements can be extracted. Slice extraction needs to freely rearrange statements, so it does not preserve control flow in general; an invariant-based presentation would hence need to consider more subtle preservation criteria.

A more straightforward extension to our current implementation would be to implement duplicate elimination as provided by Eclipse and IntelliJ. As mentioned, this extension would not seem to necessitate any extensions to the extraction mechanism proper, but only requires the implementation of a clone detection algorithm [1].

Finally, it would be interesting to see if the decomposition we have achieved for our four example refactorings could help with their verification. Since each micro-refactoring performs a very small but independent and well-defined transformation on the program it is tempting to try and verify their correctness separately, and then compose their proofs into a correctness proof of the whole refactoring. Such a proof could also make use of our previous work on formalising Java name binding [22].

8 Conclusions

We have presented a general approach to implementing software refactorings by viewing them as invariant-preserving transformations on programs in an enriched language.

Such an enriched language offers extensions to the base language that make the implementation of refactorings easier, for example by providing implicit invariant checking, or language constructs that can be used to better decompose the transformation performed by a refactoring. We have implemented a framework that offers several such extensions to the Java 5 language, and have used it to implement non-trivial refactorings such as *Extract Method*. The implementations are concise and well-structured; they support the complete Java 5 language; and our tests show that in terms of correctness they rival or even surpass widely known refactoring engines. We are confident that our approach is flexible enough to allow easy implementation of any structural refactoring that modifies name binding, control flow, and data flow.

We agree with Fowler's assessment that the *Extract Method* refactoring is a paradigmatic example of a refactoring that is simple, yet requires non-trivial analysis. But we have shown that there is no need to cross this Rubicon in one huge leap; we can instead pass it on stepping stones, one micro-refactoring at a time, with the principle of invariant preservation as our guide rope.

Acknowledgements

We would like to thank Emma Nilsson-Nyman and Ting Ting Mao, who implemented the control flow package that our refactoring implementation depends so vitally on, and Dmitry Jemerov of JetBrains, who graciously gave us access to IntelliJ's test suite. Damien Sereni provided important feedback on the ideas presented in this paper from the early stages. We thank him as well as Ran Ettinger, James Noble and the anonymous reviewers, who gave insightful comments on a draft version, which helped us improve the final paper in many ways. Finally, we would like to praise our poetic pal Pavel, without whom this paper would have a very dull title.

References

1. Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone Detection Using Abstract Syntax Trees. In: ICSM, Washington, DC, USA, p. 368. IEEE Computer Society Press, Los Alamitos (1998)
2. Eclipse (2008), `http://www.eclipse.org`
3. Ekman, T., Ettinger, R., Schäfer, M., Verbaere, M.: Refactoring bugs (2008), `http://progtools.comlab.ox.ac.uk/refactoring/bugreports`
4. Ekman, T., Hedin, G.: The JastAdd Extensible Java Compiler. In: OOPSLA (2007)
5. Ettinger, R.: Refactoring via Program Slicing and Sliding. D.Phil. thesis, Computing Laboratory, Oxford, UK (2007)
6. Ettinger, R., Verbaere, M.: Untangling: a slice extraction refactoring. In: Aspect-Oriented Software Development (AOSD), pp. 93–101 (2004)
7. Fowler, M.: Crossing Refactoring's Rubicon (2001), `http://martinfowler.com/articles/refactoringRubicon.html`
8. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification (2005)
9. Griswold, W.G.: Program Restructuring as an Aid to Software Maintenance. Ph.D. thesis, University of Washington (1991)
10. IntelliJ IDEA (2008), `http://www.jetbrains.com`

11. Johnsson, T.: Lambda Lifting: Transforming Programs to Recursive Equations. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201, pp. 190–230. Springer, Heidelberg (1985)
12. Juillerat, N., Hirsbrunner, B.: Improving Method Extraction: A Novel Approach to Data Flow Analysis Using Boolean Flags and Expressions. In: WRT (2007)
13. Kniesel, G., Koch, H.: Static Composition of Refactorings. The Science of Computer Programming 52(1-3), 9–51 (2004)
14. Komondoor, R., Horwitz, S.: Semantics-preserving Procedure Extraction. In: POPL, pp. 155–169. ACM Press, New York (2000)
15. Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., Wolfe, M.: Dependence Graphs and Compiler Optimizations. In: POPL, January 1981, pp. 207–218 (1981)
16. Mens, T., DeMeyer, S., Janssens, D.: Formalising behaviour preserving program transformations. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 286–301. Springer, Heidelberg (2002)
17. NetBeans (2008), `http://www.netbeans.com`
18. Nilsson-Nyman, E., Ekman, T., Hedin, G., Magnusson, E.: Declarative Intraprocedural Flow Analysis of Java Source Code. In: Proceedings of 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008) (2008)
19. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
20. Perera, R.: Refactoring: to the Rubicon.. and beyond! In: OOPSLA 2004: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 2–3. ACM Press, New York (2004)
21. Schäfer, M., Ekman, T., de Moor, O.: Sound and Extensible Renaming for Java. In: Kiczales, G. (ed.) OOPSLA. ACM Press, New York (2008)
22. Schäfer, M., Ekman, T., de Moor, O.: Formalising and Verifying Reference Attribute Grammars in Coq. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 143–159. Springer, Heidelberg (2009)
23. Verbaere, M., Ettinger, R., de Moor, O.: JunGL: a Scripting Language for Refactoring. In: ICSE (2006)