A Basis for AspectJ Refactoring

Shimon Rura[⋆] and Barbara Lerner

Williams College, Computer Science Department
Williamstown, MA 01267 USA
`srura@wso.williams.edu, lerner@cs.williams.edu`

Abstract. Refactorings are specific code transformations that can improve the design of existing code without changing its behavior. Many refactorings for object-oriented programs are well known, and refactoring support is now common in IDEs. Authors of AspectJ programs, however, cannot take full advantage of this wisdom and tool support for two reasons. First, there is currently a limited set of aspect-oriented refactorings. A second, more fundamental, problem is that aspect-oriented language constructs can impact what changes are behavior-preserving. As a result, even traditional refactorings are unreliable. This paper presents a framework for judging whether a program transformation is a refactoring in AspectJ. This framework is then applied to establish AspectJ-safe versions of existing refactorings and several new refactorings specific to AOP designs.

1 Introduction

A refactoring is a behavior-preserving source code transformation typically used to improve the design of a program. Refactorings have traditionally been applied *ad hoc* by programmers. Recently, however, a growing body of work has focused on identifying and describing common refactorings [Fow00] as well as providing automated support for refactoring in popular languages [Goo03]. Proponents of refactoring claim it enables better design: instead of repeatedly compromising the design in the face of changing requirements, engineers can refactor the design to more easily accommodate the code implementing the new requirements. Clear descriptions and automatic refactoring tools help software developers share useful refactorings and apply them with greater speed and fewer mistakes.

Refactoring shares its goal of enabling improved software designs with Aspect-Oriented Programming. Since most Aspect-Oriented languages are based on existing popular Object-Oriented languages, we would like to make use of OO refactorings in AOP languages. Unfortunately, the additional semantics of aspect-oriented languages make existing refactoring techniques insufficient to preserve program behavior in general. If we can understand what comprises behavior preservation in an aspect-oriented language, we can extend known refactorings so that they are valid in aspect-oriented programs, and develop new refactorings that help programmers deploy AOP features.

[⋆] Now at Kronos, Inc., Chelmsford, MA, USA

This paper provides a set of criteria that, if satisfied by an AspectJ program transformation, ensures that it will preserve program behavior. These criteria, described in the following section, are based on a system of constraints that builds on existing work in refactoring [Opd92]. While many of these constraints can be satisfied outright, some require special analyses and modifications, which are described in Section 4. Following that, we put these constraints to the test by extending known Java refactorings to AspectJ and developing new, AOP-specific refactorings.

2 An Overview of AspectJ

AspectJ [KHH+01, Tea03] is an aspect-oriented language based on Java. AspectJ supports two mechanisms for describing crosscutting features: static transformations, known as *introduction;* and dynamic invocation of code, known as *advice.* To encapsulate the various parts of a crosscutting concern, AspectJ adds to Java the `aspect` construct, which functions like a `class` but may also contain advice declarations.

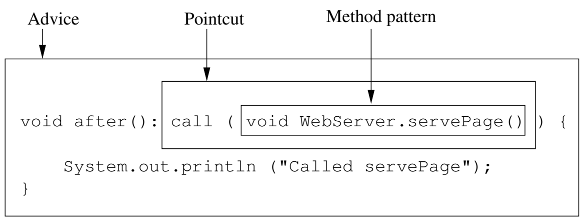
Introduction may declare new members on classes (inter-type declaration) or alter inheritance relationships. For example:

```
public int Foo.add3(int i) { return i+3; }
```

This declaration would create a method `add3(int)` as if it had been declared inside the class `Foo`, but may be written in any aspect where `Foo` is visible. Another kind of static introduction allows modifying the class hierarchy. Specifically, if a type t is declared to extend (or implement) supertype s, an aspect can declare that the type t should instead extend (or implement) the type s', provided s' is a subtype of s. (Several other static metaprogramming constructs exist in AspectJ but are not covered in this paper.)

A program's control flow can be altered using *advice,* which causes a block of code to be invoked before, after, or around (in place of) a well-defined event in program execution. These events, called *join points,* include method calls, object or class initialization, exception handling, and field reads and assignments.

Advice is parameterized with a *pointcut,* which denotes a set of join points. For example, the following advice declaration logs calls to a method `servePage()` in class `WebServer`:



```
void after(): call ( void WebServer.servePage() ) {
    System.out.println ("Called servePage");
}
```

In the example above, the advice follows all calls to a specific method. It is also possible to use *patterns* when defining a pointcut. For example, if we have several overloadings of the `servePage` method, we could log all of them with the following advice:

```
void after(): call( * WebServer.servePage(..) ) {
  System.out.println("Called servePage");
}
```

Here, the `*` matches any return type, while the `..` in the argument list matches an argument list of any length and with any parameters. The pattern mechanism is quite powerful; this example just shows some specific uses of it.

Code within the advice body may also access a variety of information about the join point, such as method arguments.

3 A Framework for Refactoring

The first major work on refactoring in object-oriented languages was William Opdyke's 1992 Ph.D. thesis [Opd92]. Opdyke presented a number of *fundamental* refactorings, argued to be behavior-preserving based on a set of constraints. He then defined more complex and task-specific *high-level* refactorings by composing fundamental refactorings in a variety of ways.

Since it is in general undecidable whether two programs have the same behavior, no set of constraints can identify *every* behavior-preserving transformation. Instead, we choose constraints that balance the flexibility to support useful changes with clearly-stated, simple requirements. The constraints we use are derived directly from Opdyke's work with updates for Java semantics. They fall into three categories:

1. **Language requirements.** A refactoring must not cause rules of language syntax and semantics to be violated. In addition to obvious considerations of syntax, a number of special situations must be avoided, such as
   - name conflicts
   - subtyping violations
   - method signature conflicts
   - type safety violations

   These particular situations are listed because they are common pitfalls of transformations involving moving, renaming, or otherwise altering the declaration of a class, method, field, or variable.
2. **Preserving subtype relationships.** We restrict our transformations to those that preserve subtyping relationships between classes and overriding relationships between methods. This guarantees that dynamic binding resolves to the same methods before and after a refactoring, limiting the effects of a change upon subtypes. In particular:
   - A class that implements any interfaces or abstract classes should continue to satisfy the requirements imposed by its interfaces or abstract superclasses after refactoring.

Note that it is possible to move a method to a superclass or subclass in some cases while honoring this constraint. For example, if a class C defines a method M and its superclass neither defines nor inherits a method with the same signature as M, moving M to the superclass still preserves subtype relationships.[1] On the other hand, if C inherits a method with the same signature as M, it cannot be moved to C because it would then change which method is dynamically bound when M is called on an object whose type is C.

To preserve subtype relationships, it is sometimes necessary to make non-local changes when carrying out the refactoring. In particular, if a method signature defined in a supertype is changed, the refactoring must also propagate this change to overriding methods in subtypes. For example, if a refactoring renames a method, all overridings of that method must also be renamed. Furthermore, the new name cannot cause any name or method signature conflicts in any subclass.

3. **Preserving semantic equivalence.** Alterations to the program must be limited to those that preserve the semantics of the altered parts. Opdyke identified the following as behavior-preserving changes:
   (a) Expressions can be simplified.
   (b) Dead (unreachable) code within a method can be removed.
   (c) Unreferenced variables, methods, and classes can be added or removed.
   (d) A variable's type can be changed, as long as each operation referenced on the variable is defined equivalently for its new type, and all assignments involving that variable remain type safe.
   (e) References to a field or method can be replaced with references to other fields or methods that are equivalently defined. If two variables are known to refer to the same object (decidable in limited cases), or if two methods have equivalent bodies (equivalent code and variable references), references to one can be replaced with references to the other.

4 Behavior Preservation in AspectJ

With aspect-oriented constructs, program elements declared outside a class can affect its structure and execution. Thus when aspects are present in a program, our refactorings must preserve these AOP semantics in ways analogous to the preserved OO semantics. We first look at the straightforward extensions of the constraints to traditional programming constructs used in aspects. Then, we examine the more interesting issue of how the new programming constructs used in AspectJ further constrain refactoring.

4.1 Effect of Aspects on Existing Constraints

Language Requirements The first consequence of allowing AOP constructs is that our language requirements are extended. The three main AspectJ features—

[1] Of course, the refactoring might fail for other reasons, such as referencing an instance variable declared in C.

inter-type member declarations, declaration of new inheritance relationships, and advice—each have several effects on language requirements.

Inter-type member declarations, which can function exactly like local declarations in a target type, can be declared in any aspect. This affects the requirements that members of a class have unique names and signatures, as both introduced and local declarations need to be checked for conflicts.

Aspects can also declare new inheritance relationships by declaring a class to implement an interface or by assigning it a new superclass. (The new superclass must be a subclass of the original superclass.) These declarations must be taken into account when changing inheritance relationships in a refactoring, as the language requirement that each class must have one direct superclass that is not its subclass may be violated. For example, a class's supertype could be changed in a way that conflicts with an introduced supertype.

Subtype Relationships Changes to methods introduced by aspects must also be analyzed with respect to their relationship to subtypes so that overriding relationships of methods are maintained. Also, if an aspect changes a class's supertype, this must be taken into consideration when evaluating refactorings that modify non-private class members to be sure that the subtyping relationships are preserved.

Semantic Equivalence Aspects provide a new programming construct whose code may be refactored and whose code must be examined for semantic equivalence. Thus, the same refactorings possible for class code are also possible for the code in aspects: expressions can be simplified, dead code can be eliminated, etc.

An additional requirement is that the code in aspects must be examined for references to classes, methods, and variables to be sure that code is unreferenced before a refactoring can delete it.

4.2 A New Constraint: Pointcut Pattern Equivalence

Advice poses a special challenge in refactoring, because, to preserve behavior, advice must apply at semantically equivalent join points before and after refactoring. Some pointcuts, such as `call`, `get`, and `set`, are statically determinable: given full source code, we can identify exactly where all appropriate method calls and field accesses occur. Other pointcuts, however, cannot generally be determined statically. These include `cflow`, `cflowbelow`, `if`, and some cases of `this`, `target`, and `args`. It seems intuitive that a refactoring should ensure each pointcut is left with either the same join points or semantically equivalent join points to those it contained before the refactoring.[2] Thus one way to argue that

[2] In some refactorings, such as *Extract Method* (Section 5.2), the locations in code that correspond to specific join points may change. In these cases it is necessary to take extra precautions to ensure that semantic equivalence can be preserved.

a refactoring is behavior-preserving with respect to advice is by showing that the meanings of pointcuts are preserved.

Another possibility, however, is to present the argument in terms of the patterns used in the pointcut expression: if the patterns match semantically equivalent program elements, then the pointcut will match semantically equivalent join points. This is a stronger requirement than simply that the pointcut contain an equivalent set of join points, because while individual patterns may each match many elements, the resulting pointcut could be small or empty. For instance, the pointcut

```
call( public int Foo.getLength() )
```

implies the following dependencies at the pattern level:

- a method called `getLength`,
- with no arguments,
- in a class called `Foo`,
- returning an `int`, and
- declared to be `public`.

If we require a refactoring to preserve the set of program elements that the type pattern matches, we will need to change the pointcut if the `public int Foo.getLength()` method is renamed. On the other hand, if we require a refactoring to preserve the set of join points, it is only necessary to change the pattern if the method is actually called.

While pattern equivalence is a stricter condition, it offers some advantages. First, the exact elements that match a pattern can always be determined statically (given complete source code), and indeed quite simply.

Second, it seems intuitive that the type pattern itself is a reference to a program element. While it is not necessary to modify the pattern to preserve the behavior of the program if the aforementioned `getLength` method is never called, it does not preserve the intentions of the programmer, who may not be certain if the method is called or not. By updating the pattern as part of the refactoring, we can preserve both semantic equivalence of the program and the programmer's intentions.

Thus, we add a fourth constraint for behavior preservation of AspectJ programs:

4. **Preserving pointcut pattern equivalence.** A refactoring must ensure that all pointcut patterns match equivalent program elements before and after the change. (This may require the pointcut patterns to be modified as part of the refactoring.)

In general, if a program element matches a pattern before applying a refactoring, it should continue to match after refactoring. If the element is changed in a way that causes it to no longer match, we must modify the pattern to ensure that it will match. Conversely, if a program element does not match a pattern before applying a refactoring, it should still not match after applying the refactoring. This may again require us to modify the pattern to avoid a new match.

There is a simple, general solution to this problem. Suppose we start with a pattern X,

– if we want to add an element, we synthesize a pattern p for that element, and replace X with $X \mid\mid p$; or
– if we want to remove an element, we synthesize a pattern p for that element, and replace X with X && $!p$.

This technique is simple—all we need is the ability to synthesize a pattern that selects exactly the element to add or remove, and the union, intersection, and negation operators on patterns. A pattern that selects exactly one element is easy to generate by using a fully-qualified name and signature.

Unfortunately, AspectJ syntax allows the set operators only within type patterns. That is, we can write

```
staticinitialization( Foo || Bar )
```

to add the type `Bar` to the type pattern `Foo`. But if we want to add a method `bar()` to the method pattern `void SomeClass.foo()`, we cannot write

```
call( void SomeClass.foo() || void SomeClass.bar() )
```

(this is a syntax error). To work around this problem, we can make the following change instead:

```
                call( void SomeClass.foo() )
                            ⇓
call( void SomeClass.foo() ) || call( void SomeClass.bar() )
```

This is an example of a general technique. That is, if we have a pattern X appearing in a pointcut pc, and we want to achieve the effect of adding or removing a certain program element e from the set matched by the pattern, we can perform one of the following replacements. To add the element matched by pattern a:

$$\mathrm{pc(\ X\ )} =: \mathrm{pc(\ X\ )} \mid\mid \mathrm{pc(\ a\ )}$$

To remove the element matched by pattern a:

$$\mathrm{pc(\ X\ )} =: \mathrm{pc(\ X\ )}\ \&\&\ !\mathrm{pc(\ a\ )}$$

In these replacements, the set operations in use apply to the pointcuts themselves, but the effect is equivalent to modifying only the pattern.

5 Existing Refactorings Revisited

We now turn our attention to existing refactorings commonly used in object-oriented programs and consider how aspects impact those refactorings. Our approach is to use the constraints described in Sections 3 and 4 to guide the identification of *preconditions* under which the refactoring may be applied and the *actions* that are taken to apply the refactoring.

5.1 Rename a Variable

Renamning a variable, although a very basic refactoring, touches upon many unique challenges that aspect-oriented languages impose on refactorings. In Java, a variable can be renamed by changing its declaration and all references. The preconditions to this renaming are:

1. The new name does not conflict with an already existing variable in the same scope (including in subclasses if the variable is an inherited field), and
2. The new name will not cause the variable to be shadowed in any place where it is currently referenced.

In AspectJ, we must introduce the following additional preconditions if the variable being renamed is a field:

3. ✂[3] The new name does not conflict with a field introduced into the same type (or its subtypes) from an aspect.

Precondition 1 ensures that language requirements are honored. Precondition 2 prevents the case where changing a variable reference to use the new name would actually cause it to refer to a new variable, thus changing program semantics.

Precondition 3 is a straightforward extension of condition 1 to encompass AspectJ language rules. Because a field introduced into a class from an aspect functions equivalently to a field declared in the class itself, introduced fields must also be checked for name conflicts.

Assuming the renaming would not violate these preconditions, we now consider the actions that would be taken to implement the refactoring:

1. The declaration of the variable and all references to the variable are changed to use the new name.
2. ✂ If a field pattern matched this field before renaming and does *not* match afterwards, it is extended to match.
3. ✂ If a field pattern did not match this field before renaming and *does* match afterwards, it is narrowed to avoid matching.

Action 1 is the normal action to apply this refactoring in a Java program. Actions 2 and 3 ensure pointcut pattern equivalence. Patterns cannot refer to local variables, so only fields are affected. Consider this example where we want to rename the field `x` to `y` in class `Foo`:

```
class Foo {
  public int x;
  public void setX(int newX) { x = newX; }
  public int getX() { return x; }
}
```

[3] The cutting scissors symbol (✂) denotes a precondition as particularly concerned with AOP behavior preservation constraints.

Suppose that the Java preconditions are met; that is, we can safely rename the declaration of x and all references. In AspectJ, however, we might have an aspect such as:

```
aspect FooListening {
  after() : set( Foo.x ) {
    // notify some listeners
    ...
  }
}
```

Before renaming, the field pattern used in the advice, `Foo.x`, referred to a field that it no longer refers to after refactoring. We want this pattern to include the field now called `y`, so according to the general technique described in Section 4.2, the pointcut becomes:

```
set ( Foo.x ) || set ( Foo.y )
```

Thus after AspectJ-safe refactoring the code is:

```
class Foo {
  public int y;
  public void setX(int newX) { y = newX; }
  public int getX() { return y; }
}

aspect FooListening {
  after() : set( Foo.x ) || set ( Foo.y ) {
    // notify some listeners
    ...
  }
}
```

In this example, the changed pointcut— `set ( Foo.x ) || set ( Foo.y )`— seems overly complex, because there is no longer a field `Foo.x`. The reason that we expand, rather than directly alter, the pointcut is because expansion is a more general solution. For example, if the pointcut had originally been `set ( *.x )`, transforming the pattern to `set ( *.y )` would change program behavior if there were any other fields in the program named `x` or `y`. In the case above, simplifying the pointcut to `set ( Foo.y )` can be considered an additional refactoring.

5.2 Extract Method

The Extract Method refactoring entails replacing lists of statements that occur repeatedly in a program with method calls. It is an example of a *high-level* refactoring, one that composes a number of other refactorings to accomplish a more specialized purpose. The two refactorings being composed first create a new method and then replace one or more statement lists with a call to the new method. For example, suppose our class contains several lines of code that are used to reinitialize instance variables in several places:

```
  public void m () {
    count = 0;
    list = new Vector();
    ...
  }
```

The *Extract Method* refactoring allows us to define a new method, `init`, with the repeated statements as its body and replace those statements with a method call, so we get:

```
  private void init() {
    count = 0;
    list = new Vector();
  }

  public void m() {
    init();
    ...
  }
```

As in plain Java, the method we create as part of *Extract Method* contains the same statements we want to extract, and accepts any internally referenced variables not in the scope of the method declaration as parameters. Similarly, the statement lists are replaced with method calls in the same way as in plain Java. However, aspects affect the refactoring by imposing additional preconditions on when the refactoring can be applied. In particular, we need to ensure that any advice that applied to the inline statements applies equivalently to the statements in the extracted method. We give an example here and a more precise description of these preconditions below. Suppose that we have advice using the following pointcut:

```
  set ( Foo.count )
```

In this case, the refactoring shown above would be allowed since the setting of the `count` variable served as a join point both in the statement list and in the method. Now, consider a slightly different pointcut:

```
set ( Foo.count ) && within ( Foo.m () )
```

In this case, we would want to prohibit the replacement of the statements in `m` with a method call to `init` because the assignment within `init` is not in the pointcut while the assignment in `m` is. Allowing the refactoring would cause the advice to no longer be invoked where it once was, thereby causing a change in behavior.

Since this is a high-level refactoring, we analyze it by examining the component refactorings. The preconditions for the Create Method refactoring are:

1. The lines of code change at most one local variable.
2. The new method will compile as a member of the target class.
3. If the new method will overload an existing method (either in the target class, in its subclasses, or one statically introduced into the target class or subclasses), it must either be more general (thus not capturing any calls that would have invoked the existing method) or more precise and semantically equivalent (e.g. identical) to the method it overloads.
4. If the target class has an inherited method that will be overridden by the new method, either that method is unreferenced on the target class and its subclasses, or the new method is semantically equivalent to the method it overrides[4]. Changes to the class hierarchy caused by aspects must be taken into consideration as well when evaluating this precondition.

The first precondition guarantees that we can create the method. The second precondition implicitly guarantees no signature conflicts with current locally declared or introduced members of the class. The third precondition guarantees valid overloading. The final precondition guarantees that even if the new method overrides an inherited method, program behavior is preserved.

After the *Create Method* refactoring is applied, the next step in *Extract Method* is application of *Replace Statement List with Method Call*. Here we replace a statement list L in a method m with a method call mc. The following preconditions are required:

1. The called method, mc, is visible from the calling method, m.
2. If mc is not private, all subclasses that inherit m also inherit mc or the method that overrides mc is semantically equivalent to mc.
3. The call to mc is semantically equivalent to L, i.e. their abstract syntax trees are the same, up to variable renaming, and they reference semantically equivalent items outside their scopes.
4. ✂ The method mc is not matched by any method patterns. (This precludes new invocation of advice parameterized by `call`, `execution`, and `withincode` pointcuts.)
5. ✂ If any statement in L includes an assignment to or read from a field matched by a field pattern used in a `set` or `get` pointcut, that pointcut is not intersected with a `within` or `withincode` pointcut.
6. ✂ If any statement in L includes a method or constructor call matched by a method or constructor pattern used in a `call` or `execution` pointcut, that pointcut is not intersected with a `within` or `withincode` pointcut.

The first three preconditions ensure that the called method will behave equivalently to the statements it replaces. The rest of the preconditions are AspectJ-specific, and guarantee that this refactoring does not change which points in program execution cause advice to be invoked. To preclude invoking advice on

[4] It is not always possible to determine whether a particular method in a given class is referenced because of dynamic method dispatch. In many cases, however, it is possible to determine conservatively that a method is not referenced.

mc itself, we require that it is not matched by any method patterns. Further, we do not want our use of the method call to cause invocation of any advice not already invoked in our statement list. Similarly, we do not want our use of the method call to preclude invocation of advice that *was* invoked by a statement in the list. The only way, aside from the dynamically-determined `cflow` and `cflowbelow` pointcuts, to achieve this kind of limitation in a pointcut is to use `within` or `withincode` pointcuts. By intersecting (&&) these with other pointcuts, it is possible to constrain advice executions to join points occurring in a specific set of types or methods. We cannot, in general, maintain semantic equivalence of pointcuts that use `within` or `withincode` by generalizing or restricting the `within` clauses. For example, if we attempted that with our earlier example, we would end up with a pointcut:

```
set ( Foo.count ) && ( within ( Foo.m () || within ( Foo.init () ) )
```

This would be a valid refactoring of the pointcut only if the `init` method was only called from `m`. Rather than creating more complicated preconditions and defining the corresponding actions to fix these pointcuts to include or exclude specific join points, we simply require as a precondition that they do not apply. In general, refactorings that change the static location of a join point cannot be applied if a pointcut depends on these static locations by using `within` or `withincode` expressions.

The actions taken by this refactoring are the same as for the comparable object-oriented refactoring and involve creating the method, moving the statements into the method, introducing necessary local variable declarations and a return statement if necessary, and replacing the original statements with a call to the method. No additional actions are required due to aspects.

6 New AOP-Specific Refactorings

Now, we examine some refactorings that would be useful for aspect-oriented programming.

6.1 Move Local Member Declaration to Aspect

This simple refactoring removes a field or method declaration from a class or interface and moves that declaration into an aspect. From the aspect, introduction is used to add the member into the class it was taken from. For example, a method `public String toString() { ... }` in a class `Foo` can be declared from within any aspect as:

```
  public String Foo.toString { ... }
```

The refactoring has only one precondition:

1. The member must be public.

Since the introduction is a static transformation, the introduced declaration is always legal if the local member declaration is legal. The only additional requirement, then, is that the introduction is possible. This necessitates that the member be declared `public`, because introduction of a `protected` member is not allowed by AspectJ and an introduced member declared `private` would mean private to the containing aspect, not to the target class.

The action is to remove the declaration from its class and move it to the desired aspect. The name of the item must be qualified with the class name.

6.2 Generalize `before` or `after` advice to `around` Advice

If a programmer wants to declare advice behavior around a pointcut currently referenced in `before` or `after` advice, it is easy to change the advice declaration into an equivalent `around` advice. The action is to change the advice type to `void around` and

– in the case of `before` advice, add a `proceed()` statement at the end of the advice body;
– in the case of `after` advice, add a `proceed()` statement at the beginning of the advice body.

There are no preconditions for this refactoring.

6.3 Extract Disjoint State into Aspect

Like *Extract Method,* this a *high-level* refactoring which composes a number of other refactorings to accomplish a more specialized purpose. In this case, we apply the *Move Local Member Declaration to Aspect* refactoring to several members in order to separate out the declarations of peripheral properties of the class. If the set of members of the class is M, we first select a set of members $D \subseteq M$ such that:

$$\forall d \in D : \neg\text{referencedFrom}(M \setminus D, d)$$

and

$$\forall m \in M \setminus D : \neg\text{referencedFrom}(D, m)$$

where referencedFrom(X, y) is true if and only if a member in the set X references the member y.

In short, we choose a set of members that neither refers to nor is referenced from the remaining members. Although M is such a set, the intention is of course to move a proper subset of the class members, not all of the class members.

Example here: memoization won't work; it's not disjoint. This is really something where two unrelated things are stuck into one class. Why would this be? I have to think about it some more.

6.4 Extract Interface Implementation into Aspect

The purpose of the Extract Interface Implementation into Aspect refactoring is to create a new aspect that contains the variables and methods that a class contains for the sole purpose of implementing a particular interface. Since this refactoring involves only moving parts of a class to an aspect for static introduction, there are no preconditions.

The actions taken by this refactoring are more interesting. Given the name of the interface whose implementation should be extracted, the actions involve:

1. Moving the members that implement the interface to the new aspect. We'll call this set of members M.
2. Moving additional members of the class that are referenced by members in M but are not referenced by members outside of M and are not members of the class to satisfy some other subtyping constraint. As more members are moved to the aspect, we also examine which members they reference.
3. Removing the `implements` clause from the class declaration.
4. Adding a `declare parents` statement to the aspect to maintain the subtyping relationship.

The work on creating refactorings that are specific to aspect-oriented programming is just beginning. A tool that offered a rich set of such refactorings would make an excellent platform for transitioning existing Java programs into AspectJ.

7 Related Work

Exploring the interactions between AOP and refactoring is a relatively young research area that has recently become a priority for AspectJ [Kic01]. Hanenberg et al. [HOU03] offer several updated OO refactorings and new AOP-specific refactorings that are not considered in this paper. They are also developing a refactoring tool for AspectJ. Their transformation constraints adopt only the more general requirement that pointcut meanings are preserved, rather than exploiting the pattern technique described here.

Hanneman, Fritz, and Murphy [HFM03] are also examining refactoring for AspectJ. While our goal has been to provide behavior-preserving transformations, as is the goal in an object-oriented language, their goal is to maintain programmer intentions rather than necessarily preserving behavior. The following example highlights the difference. Suppose there is a pointcut defined as `call (public * *(..))`. This would match all public method calls. Now suppose a public method is refactored to be private. To preserve behavior, we would claim that one effect of the refactoring is to change this pointcut to && in the method that just became private. Hanneman's approach says that while this may preserve behavior it might violate the intention of the aspect since the aspect's purpose may be to only apply to public method calls, for example. Of course, it is not possible, in general, to discern the programmer's intent. Hanneman's solution is to engage the user in an interactive dialogue to determine their intent. This raises very interesting issues. Intuitively, it seems that when a pointcut refers to a specific program element, it is probably the programmer's intent to also modify the pointcut when a refactoring affects the program element. On the other hand, when the pointcut uses an expression with wildcards, as the example above, it seems more likely that the refactoring should not preserve pointcut equivalence to capture the programmer's intent, thus violating behavior preservation.

8 Conclusions and Future Work

The framework presented here forms a basis for judging whether proposed refactorings are behavior-preserving in AspectJ. It is the basis for the refactorings presented in Sections 5 and 6 as well as additional refactorings we have explored [Rur03]. While we have found it helpful in formulating refactorings, a major area of future work is to build a refactoring tool that can apply these refactorings to AspectJ programs to discover when they facilitate the migration from pure Java to AspectJ and to what extent they provide the type of refactoring support an AspectJ programmer most needs.

References

[Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 2000.

[Goo03] Google Web Directory. Refactoring Tools. Website, 2003. Available at `http://directory.google.com/Top/Computers/Programming/Methodologies/Refactoring/Tools/`.

[HFM03] Jan Hannemann, Thomas Fritz, and Gail C. Murphy. Refactoring to aspects — an interactive approach. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology Exchange*, October 2003.

[HOU03] Stefan Hanenberg, Christian Oberschulte, and Rainer Unland. Refactoring of aspect-oriented software. In *Net.Objectdays*, Erfurt, Germany, September 2003.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, Budapest, Hungary, 2001. Springer. Available from `http://eclipse.org/aspectj/`.

[Kic01] Gregor Kiczales. Aspect-oriented programming–the fun has just begun. In *Workshop on New Visions for Software Design and Productivity: Research and Applications*, Vanderbilt University, Nashville, Tennessee, December 13-14 2001.

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Rur03] Shimon Rura. Refactoring aspect-oriented software. Undergraduate Thesis, Williams College, 2003.

[Tea03] The AspectJ Team. *The AspectJ Programming Guide*, 2003. Available from `http://eclipse.org/aspectj/`.