

Refactoring of Aspect-Oriented Software

Stefan Hanenberg, Christian Oberschulte, Rainer Unland

University of Duisburg-Essen,
Institute for Computer Science and Business Information Systems (ICB)
45127 Essen, Germany
{cobersch|shanenbe|unlandr}@cs.uni-essen.de

Abstract. The application of refactorings during an object-oriented development process improves the design and therefore the quality of software. Aspect-orientation is a new programming paradigm that increases the modularity of software. Hence, it seems natural to apply both aspect-orientation as well as refactoring during a software development process since both techniques permit to increase the modularity and comprehensibility of software. However, on the one hand existing object-oriented refactoring techniques cannot directly be applied to aspect-oriented software because traditional object-oriented refactoring applied in an aspect-oriented environment is no longer behavior preserving. On the other hand, since the new features of aspect-oriented languages permit to modularize software in different ways there is a large variety of new refactorings based on these features. This paper discusses the relationship between object-oriented refactoring and aspect-orientation. We show what aspect-oriented elements conflict with existing refactorings and propose solutions for these conflicts for the aspect language AspectJ. Furthermore, we introduce a number of new aspect-oriented refactorings which help on the one hand to migrate from object-oriented to aspect-oriented software and on the other hand to restructure existing aspect-oriented code.

1 Introduction

Aspect-Oriented Programming (AOP, [9]) provides means to encapsulate concerns which cannot be modularized using traditional programming techniques. These concerns are called *crosscutting concerns*. Prominent examples for such concerns are tracing, concurrency control or transaction management. Aspect-oriented programming techniques provide two new concepts: *join points* and *aspects*. Join points specify the elements to be modified by aspects. Aspects encapsulate the implementation of crosscutting concerns and refer to a number of join points. Aspect-orientation is based on *weaving*: a weaver integrates a number of aspects into a base system on the basis of given join point definitions.

Meanwhile a number of aspect-oriented programming techniques are publicized like *AspectJ* [1, 10], *Sally* [7], *PROSE* [13, 14] or *AspectS* [8] which are either extensions of an object-oriented base language or frameworks written in such a base language. These techniques provide features for specifying join points and aspects. As-

pectJ, which is an extension of the programming language Java, is the most popular one and already has a large community.

Refactoring [3,15] is a technique to restructure object-oriented code in a disciplined way. Refactorings are *behavior preserving program transformations* [16]. The intention of refactoring is to improve the readability and comprehensibility of object-oriented code. For an efficient application of refactoring tool-support is needed. Nowadays, refactoring tools are available for a large number of integrated development environments. Such tool support is also a precondition for the successful application of lightweight development processes like *eXtreme Programming* (XP, [2]). It uses refactoring as one of its key components.

Most refactorings increase the modularity of code and eliminate redundancies. Since the same advantages are gained by aspect-oriented software development it seems to be natural to apply refactoring and aspect-oriented programming within the same development process.

The benefit of using both approaches is threefold. First, refactoring can be used to restructure the base program to which aspects are woven. This increases the comprehensibility of the base program without the need to understand the woven aspects. Second, refactoring can help to restructure object-oriented code in an aspect-oriented way. This permits to migrate from object-oriented to aspect-orientated software. And third, refactoring can be applied to the aspect-oriented constructs to increase their comprehensibility and modularity.

Transforming an application inevitably leads to a modification of join points to which aspects might be woven to. Consequently, if aspects are not aware of those modifications and their join point specification is not adapted refactorings are no longer behavior preserving. As a result, current refactoring tools cannot be used to refactor the object-oriented base system. Furthermore, refactorings are needed which make use of aspect-oriented features to restructure software along those new features.

Hence, for the application of refactoring and aspect-oriented techniques within the same software project refactoring needs to become *aspect aware*. This paper proposes how to make refactorings aspect aware. That means we propose modifications of existing object-oriented refactorings to be used in conjunction with aspect-oriented techniques. Furthermore, we propose a number of new refactorings which already make use of aspect-oriented features.

In section 2 we give a small example which reveals the need for aspect-aware refactorings. Afterwards, we discuss in detail the interplay of object-oriented refactorings and aspect-oriented techniques and the resulting conflicts. In Section 4 we propose aspect-aware modifications of some well-known object-oriented refactorings taken from the catalogue in [3]. Then, we propose a number of often used aspect-oriented refactorings. In section 6 we present a tool we developed that supports AspectJ aware refactorings in Eclipse. Finally, we conclude the paper.

2 Example

Figure 1 shows a class `TemperatureSensor` from an AspectJ environment that represents a temperature and humidity sensor. The physical temperature sensor sends

the current temperature every ten milliseconds to an instance of TemperatureSensor using the method `setTemperature`. Every second the physical humidity sensor sends the current humidity using `setHumidity`.

```

class TemperatureSensor {
    List _temp = new ArrayList();
    double _humidity;
    double getHumidity() { return _humidity;}
    void setHumidity(double hum) {
        _humidity = hum;
        Display.updateDisplay(hum);
        this.doInsert(hum);
    }
    double getAverageTemp() {Iterator it = _temp.iterator ...}
    void setTemperature(double temp) {
        _temp.add(new Double(temp));
        Display.updateDisplay(temp);
        this.doInsert(temp);
    }
    public void doInsert(double value) {
        //open database connection, insert new value, etc.
        ...
    }
}
public class Display {
    static void updateDisplay(double value) {
        ...
    }
}

```

Fig. 1. Temperature and Humidity Sensor

Whenever an application requests the current temperature and the current humidity the last value of humidity is taken and the temperature of the last second is computed. The latter means that the average temperature of the last hundred values is computed (`getAverageHumidity()`). All temperature and humidity values are stored in a database. Furthermore, a class `Display` is to be informed whenever a new value for the temperature or humidity is stored. This implies to call the method `doInsert` from within `setTemperature` and `setHumidity`. `doInsert` writes the new values into a database. The display is informed about new values by calling the (static) method `updateDisplay`.

Although the class works in the required way there are a number of disturbing elements. They are responsible for the *bad smell* [3] which should drive the developer to refactor the class. The first thing is, that the method `doInsert` belongs to the persistency concern and somehow does not directly belong to the `TemperatureSensor`. Hence, we create a new aspect `PersistentTemperatureSensor` which introduces this method. Furthermore, we recognize that inside `setHumidity` and `setAverageTemp` the calls for writing the new values into the database are redundant.

Hence, we move the statements to a piece of advice and specify a corresponding pointcut (Figure 2).¹

```
class TemperatureSensor {
    List _temp = new ArrayList();
    double _humidity;
    double getHumidity() { return _humidity;}
    void setHumidity(double hum) {
        _humidity = hum;
    }
    double getAverageTemp() {Iterator it =_temp.iterator ...}
    void setTemperature(double temp) {
        _temp. add(new Double(temp));
    }
}

public aspect PersistentTemperatureSensor {
    public void TemperatureSensor.doInsert(double value) {
        //store value in db
    }
    pointcut setter(TemperatureSensor ts, double value):
        target(ts) && args(value) && (
            call(void TemperatureSensor.set*(double)));
    after(TemperatureSensor ts, double v): setter(ts, v) {
        Display.updateDisplay(v);
        this.doInsert(v);
    }
}

public class Display {
    static void updateDisplay(double value) {
        ...
    }
}
```

Fig. 2. First Refactoring of TemperatureSensor

Another somehow disturbing thing is the name of the method `setTemperature`. Since the temperature is not only a single value but a list of values, it seems to be worthwhile to rename this method to `addTemperature`. Hence, we apply the *rename method refactoring* [3, p. 273]. However, just renaming the method and its calls does not lead to a behavior preserving application. Because of the aspect we also have to modify the pointcut definition in `PersistentTemperatureSensor`.

The modification of the pointcut is not trivial: we used a wildcard within the pointcut definition which we cannot use any longer since the methods' names do not start with the same characters any longer. The most obvious modification of this pointcut is that we use two pointcut designators, one for each method.

```
pointcut setter(TemperatureSensor ts, double value):
    target(ts) && args(value) && (
        call(void TemperatureSensor.setHumidity(double))
        call(void TemperatureSensor.addTemperature(double)));
```

¹ Note, that we performed in these last two steps behavior preserving transformations by shifting object-oriented code to aspect-oriented constructs.

We consider `TemperatureSensor` once more and realize that the class not only cares for the temperature but also for the humidity and thus represents two different sensors at the same time. From that point of view it is appropriate to design `TemperatureSensor` as a class `TemperatureHumidityController` which owns two different objects: a `TemperatureSensor` and a `HumiditySensor`. Hence, we rename `TemperatureSensor` and apply the *extract class refactoring* [3, p.149] two times, one time to extract `TemperatureSensor` and another time to extract `HumiditySensor`.

However, the problem with the aspect `PersistentTemperatureSensor` becomes even worse. First, we need to adapt the type pattern in the introduction since we now need to introduce `doInsert` to two different classes:

```
public void (TemperatureSensor || HumiditySensor).
doInsert(double v) {...}
```

Furthermore, we have to adapt the pointcut once more. The problem here is now, that the pointcut originally passed the target object as a parameter to the piece of advice. But the situation now is that the pointcut is related to different classes which do not have any common superclass (except `Object`). Hence, first of all it is necessary to restructure the original pointcut before thinking about how to adapt the pointcut according to this transformation.

From the example the following can be learned: first of all we need some well-defined rules about how to transform object-oriented constructs into aspect-oriented ones. Then we realize that the application of pure object-oriented refactoring no longer leads to behavior equivalent applications. And finally, we identified the need for refactoring aspect-oriented code.

3 Join Point Specification and Refactoring

The reason for the conflicts between object-oriented refactoring and the aspect-oriented features lies in the aspect specification. Aspects are woven with the help of certain join points defined by the *join point language* which is the fundament of each aspect-oriented technique. Hence, it is necessary to understand in detail how join points are specified before it is possible to determine (and solve) the conflicts between object-oriented refactorings and aspect-oriented features.

The join point language specifies those elements inside the base program to which aspects are to be woven to. It depends on the underlying aspect-oriented techniques how join points are identified².

In AspectJ join points for introductions are specified by *type patterns* which determine the target types to which additional members are to be added. Join points to change the runtime behavior of a system are specified by a *pointcut definition*. The join point specification in AspectJ is mainly based on the model of *lexical join points*

² For example in *Sally* [7] join points are specified by queries on source code of the base program. In *AspectS* [8] a single join point consists of a class descriptor and a method selector.

[12, 11]: join points inside the base program are identified because of a lexical correspondence. For example, a type pattern like `Temp*` determines all types which begin with the characters `Temp`. In the same way a pointcut designator's parameter is lexically compared to the base program's sources. For example a pointcut "`call(void TemperatureSensor.set*(double))`" specifies all calls of methods that begin with the characters `set` and contain a parameter of type `double` on an instance of a class with the name `TemperatureSensor`³. Method calls can be directly extracted from the base program's abstract syntax tree. For the target class and the parameter types additional informations from the type system are needed. Such informations are extracted by a static analysis.

Furthermore, AspectJ provides pointcut designators for *dynamic join points*. Dynamic join points depend on run-time information which cannot be determined by a static analysis. For example the `args` pointcut designator chooses method or constructor calls whose actual parameters have a certain type⁴.

Finally, AspectJ permits to specify *structural join points*. Such join points are chosen because of a certain context. For example, the `within` pointcut designator chooses all join points within a certain class (described by a type pattern). In the same way the `cflow` pointcut designator selects those (dynamic) join points which occur in the control flow initiated by another join point.

Aspects are woven to a set of join points provided by the base program. Since refactorings are transformations of the base program they change the set of join points. For example, *rename method* as shown in the introducing example changes a lexical join point. That means that all join points in an aspect specification which lexically refer to the method that is about to be renamed no longer refer to the same join point after the refactoring.

Most refactorings (like *extract class*) change the structure of the program. Hence, those refactorings conflict with structural join points. Almost every pointcut specification in AspectJ applications contains structural join points. Even a `call` pointcut like "`call(void TemperatureSensor.setTemperature(double))`" contains structural elements because it refers to the method `setTemperature` defined in class `TemperatureSensor`. Hence, refactorings like *push up method* or *push down method* [3, p. 328] which move a method to a super- or a subclass conflict with these structural join points since the type pattern "`TemperatureSensor`" is no longer valid.

Therefore, when applying a certain refactoring to an application the affected join points must be determined. For this, the join point language needs to be analyzed with respect to what constructs are used to determine what kinds of join points. In an AspectJ environment it needs to be analyzed what pointcut designators are used to determine those affected join points. Then, we need to determine all aspects which make use of those constructs since those aspects potentially lead to some conflicts. Then, it needs to be analyzed what existing aspects refer to the join points which are about to be affected.

Then we need to determine how the join points are affected by the refactoring. On the one hand the *quantity* of join points can be affected by a refactoring. For example

³ Note that `TemperatureSensor` and `double` are just type patterns.

⁴ Which is again lexically checked by a type pattern.

extract class [3, p. 149] creates a new class which provides a number of new join points while *inline method* [3, p. 117] reduced the number of join points. On the other hand the *quality* of join points can be affected. For example *remove parameter* [3, p. 277] does not decrease the number of join points, but reduces the ability for an exact definition of where an aspect should be woven to (e.g. in cases where aspects are woven to messages of a certain parameter type list).

Then it needs to be determined whether those aspects that potentially lead to some conflicts are influenced by the way how the join points are affected. That means it must be determined whether the way how they are woven to the application is changed. In such a case strategies need to be determined that modify the corresponding join point specification.

Hence, it is necessary to analyze for each refactoring its effect on the join points to provide strategies to solve conflicts in the aspect specification.

4 Aspect-aware Refactorings

Aspect-aware refactorings are refactorings that can be applied to the base program of an aspect-oriented system. In order to provide behavior preservation, aspect-aware refactorings not only transform the base program but also all necessary join point specifications.

In [15] Opdyke proposes a set of enabling conditions to preserve the observable behavior. We suggest the following additional enabling conditions to ensure that behavior is preserved by refactorings in aspect-oriented system.

1. The quantity of those join points which are addressed by a particular pointcut is not changed after refactoring.
2. Those join points which are addressed by a particular pointcut have an equivalent position within the program control flow in comparison to the state before refactoring.
3. The join point information offered by each pointcut does not decrease.

The first condition describes that whenever a transformation changes the base program of an aspect-oriented system and thereby increases or decreases the number of available join points each pointcut needs to address the same number of join points as before. The second condition means, that each join point occurs at the same position. The last condition says that the information available at a join point should not decrease. This becomes necessary if an aspect's behavior is supposed to vary depending on the information offered by its pointcut.

In this section we introduce aspect-aware versions of *rename method* [3, p. 273], *extract method* [3, p. 110] and *move method* [3, p. 217], because on the one hand they are supported by most refactoring tools and on the other hand those refactorings conflict with different kinds of join points.

We concentrate in this section on the impact of known refactorings on the join point specification in AspectJ and describe what further needs to be considered to preserve behavior. It is not our intention to repeat the well-known proceedings of each refactoring in detail. Such descriptions can be found for example in [3].

4.1 Rename Method

Sometimes a method name is not chosen adequately in the first attempt or the name is no longer appropriate since the implementation changed. In such situations *rename method* [3, p. 273] is to be applied.

```
class C {
    void foo() {...}
}
..
pointcut pc1():
    call(void *.foo());
..
pointcut pc2():
    call(void *.bar());

class C {
    void bar() {...}
}
pointcut pc1():
    call(void *.foo()) ||
    call(void C.bar());
pointcut pc2():
    call(void *.bar()) &&
    !call(void C.bar());
```

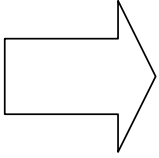


Fig. 3. Renaming a method `foo()` with pointcuts `pc1` and `pc2`.

We have to consider all pointcut designators that deal with signature patterns, because renaming a method means to change a signature. Such designators are `call`, `execution` and `withincode`.

To make rename method aspect-aware we need to do the following additional steps. First, we have to identify all pointcuts related to this signature using the above mentioned designators. Second, we rename the method and rename all method calls to it, too⁵. The further proceeding at each pointcut depends on whether each one addresses just one single join point or a collection of join points.

If the pointcut addresses just a single join point we simply propose to replace the signature pattern. If a pointcut refers to a collection of join points (caused e.g. by wildcard utilization, see Figure 3) the following steps are to be applied:

- If the pointcut is anonym, create a named one. If it is a multifid pointcut we apply *Composite Pointcut* as described in [4].
- Copy the affected pointcut designator and replace the signature pattern according to the new method name.
- Compose the pointcuts using the `||` operator.

This approach maintains the original set of join points addressed by the pointcut (see `pc1` in Figure 3). Hence, we fulfill the first enabling condition. Finally, we have to check whether the new method signature is referred by a pointcut that did not refer to it before. In such a case we propose the following steps to remove the method from the focus of that pointcut:

- If the pointcut is anonym, create a named one. In the case of a multifid pointcut it is useful to apply *Composite Pointcut* as described in [4].

⁵ Take into account that the aspect code may contain calls to the relevant method which need to be renamed, too.

- Create a new named pointcut from a copy of the existing one. Replace the signature pattern with the new method signature and negate it by using the operator `!`.
- Compose the pointcuts using the `&&` operator.

The negation excludes the join point described by the method from the collection of join points addressed by the pointcut `pc2`. Thereby the first enabling condition is fulfilled. Note that the further enabling conditions are not violated by *rename method*.

4.2 Extract Method

Extract Method [3, p. 110] is usually applied to decrease the complexity of the method and to increase the reusability of the extracted code. As a result a new method and a corresponding method call are created.

Naturally the number of join points is increased by adding a new method. This method represents a new node into the run-time object call graph of the program and can be affected by `call` and `execution` pointcuts. Since the number of join points is increased inside a class additional pointcut designators of interest are `within` and `withincode` (because of changed control flow inside the original method).

The newly created method signature could violate the first enabling condition. In the previous section, we already described a suitable countermeasure that is able to fulfill the condition. However, `within` depicts an exception. This designator picks all available join points inside a class designated by its type pattern. We can avoid the violation by negating the additional call and execution join points.

A second pitfall depicts the alteration of control flow within the initial method. By performing the transformation, one or more statements will be extracted into a new method and a reference to that is added. Assuming the new signature is addressed by a signature pattern used in `withincode`, the pointcut's collection of actual matched join points changes. We have discovered that the following proceeding is appropriate to avoid that pitfall (see also the example from Figure):

- If the initial pointcut is anonym, create a named one. If the pointcut is multifid, use the *Composite Pointcut* strategy.
- Build a call pointcut that refers to the extracted method signature, negate it and compose it with the initial pointcut using the `&&` operator. After that enclose the outcome in round brackets.
- Create a `cflow` pointcut that comprises a `withincode` pointcut whose signature pattern refers to the initial method we had extracted the code fragments from. This `cflow` pointcut causes that solely invocations of `C.bar` inside of `C.foo` adds join points to the collection of pointcut `x`.
- Create a second `withincode` pointcut referred to the extracted method.
- Build a negated execution pointcut that refers to the extracted method signature, compose these last three described pointcuts using the `&&` operator and enclose it in round brackets.
- Finally, compose both bracketed pointcuts using the `||` operator.

This proceeding maintains the pointcut's set of actual picked join points. All join point positions within the program control flow are equivalent. Even if join point information is not used in this example, note that all relevant join points possess the same kind (set, call, etc.) than before the transformation and the information supply at the join points is not decreased.

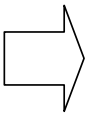
<pre>class C { void foo(int a, int b){ goo(); System.out.println(a + b); this.x = a; }... } pointcut x(): withincode(void *.foo(..));</pre>		<pre>class C void foo(int a, int b){ goo(); bar(a, b); } void bar(int a, int b){ System.out.println(a+b); this.x = a; } ... } pointcut x(): (withincode(void *.foo(..)) && !call(void C.bar(int, int))) (withincode(void C.bar(int, int))&& !execution(void C.bar(int, int))&& cflow(withincode(void C.foo(..))));</pre>
--	---	--

Fig. 4. Extract method bar from foo.

4.3 Move Method

Move method [3, p. 142] is applied if the method rather belongs to a different class than where it is currently defined.


<pre>class C { void foo() {...} ... } pointcut x(C c): execution(void *.foo()) && this(c);</pre>		<pre>class D { void foo() {...} ... } pointcut x(C c): execution(void *.foo()) && this(c); pointcut y(D d): execution(void D.foo()) && this(d);</pre>
---	---	---

Fig. 5. Move method example with context exposure

We have to consider that in consequence of the moving action the type patterns might not address the new destination type. Hence, the pointcut designators of interest are all designators which contain a type pattern. Such designators are in addition to the above explained ones *target* and *this*.

If the method we want to move to another class is already referred by a pointcut the further proceeding depends on whether the pointcut exposes parts of the execution context by applying *target* or *this*. If the relevant pointcut does not capture context at the join point, the following proceeding is suitable:

- If the relevant pointcut is anonym create a named one.
- Copy the affected pointcut and replace the type pattern within the signature pattern with the new type.
- Compose both pointcuts using the `||` operator in a third named pointcut.

Assuming that the pointcut deals with context exposure, we propose the following steps to maintain the relation (see also Figure 5):

- Copy the affected pointcut and create a new named pointcut. Replace the type pattern within the signature pattern by the new type.
- Change the object type of the pointcut arguments used by the `target` or `this` pointcut.
- Copy the pieces of advice applied by the initial pointcut and replace the corresponding pointcut.
- Check whether the pieces of advice are still type correct in respect to the changed type. If they are not, consider introducing or moving further elements.

It is not possible to make use of a composite pointcut due to the varied execution contexts. These steps preserve the first enabling condition. Note that the relevant join point information supply is not decreased.

5 Aspect-Oriented Refactorings

In contrast to the previously described refactorings the intention of aspect-oriented refactorings is to restructure the underlying object-oriented base program using AO language features or transforming pure aspect code.

5.1 Extract Advice

It is often observable that different methods scattered throughout the program have calls to comparable methods or include similar behaviour that cannot be factored out straightforward into method declarations. Often such a situation is given if *extract method* has been applied for several times and the extracted method is often reused. That means although redundant code is already extracted into a method, there are still a number of redundant method calls. In such a situation the redundant code is to be extracted into a piece of advice. This would modularize the redundant parts and coincidentally spreads out the behaviour over the program.

The participants of the refactoring are the code fragment we want to factor out, the signature of the method we want to extract from, the pointcut that should pick out the join point and the piece of advice.

The hardest problem of *Extract Advice* is to deal with local and temporary variables. Hence, *Extract Advice* is restricted in many ways. Because of the AspectJ inherent limitations of context exposure, it is not possible to expose local variables from the execution context. Method arguments depict an exception. Nevertheless, we are able to factor out local variables as well, if the variable declaration is part of the se-

lected code fragment and the remaining method control flow is not including further applications of the variable.

```

class C extends B {
    static void goo(int a, int b){
        ...
    }
    void foo(int a, int b){
        goo(a, b);
        this.bar();
        System.out.println(a + b);
    }
    ...
}

class C extends B {
    static void goo(int a,int b){
        ...
    }
    void foo(int a, int b){
        System.out.println(a + b);
    }...
}

aspect A {
    pointcut x(C c, int a ,int b):
        execution(void C.foo(int,int))
        && target(c) && args(a, b);

    around(C c, int a,int b): x(c,a,b){
        C.goo(a, b);
        c.bar();
        proceed(c, a, b);
    }...
}

```

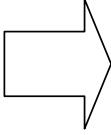


Fig. 6. Extract advice example

A second restriction depicts method calls to private members. We cannot gain access to private members within an advice, except we declare the aspect in which the advice is located as `privileged`.

Furthermore, we have detected a restriction at applying the keywords `this` and `super` in an extracted piece of advice. In the body of an advice `this` and `super` refer to the instance of the aspect. An object-identifier of the `target` or `this` pointcut can substitute the keyword `this`.

The first step to extract an advice is to create a pointcut that targets at the relevant method. In case the method is non-static we have to utilize context exposure, which means we have to pass the caller and callee as a parameter.

The second step is to make a decision on what kind of piece of advice the refactoring is to be applied. Depending on the position of the code fragment within the initial methods and the availability of a return type, we have to choose an appropriate kind of advice. It is recommended to use the `around` advice. The `around` advice substitutes the join point rather than running before or after it. Further advantageous properties are `around`'s capability to return a value and the ability to execute the computation of the original join point with the `proceed` form. `Proceed` takes as arguments the context exposed by the pointcut and returns the same type as declared by the piece of advice. Conditioned by the code fragment position `proceed` has to be placed as first or last statement in the advice body.

The third step is to copy the selected code fragment into the advice body and transform all member calls. Static calls are to be extended with their type names and member calls with the context exposing parameter of a `target` or `this` pointcut. Finally, we are able to remove the code fragment from the initial method.

This refactoring also suffers from the `withincode` problem as already mentioned in *Extract Method*. If the join points within the original method body were affected by a pointcut previously, we can not perform the code transformation without violating the enabling conditions.

5.2 Extract Introduction

Often class definitions contain members which are part of the concern the classes were originally defined for and members which are part of the implementation of further features, which need to be added to the class, but which do not directly belong to the original intention. *Extract introduction* removes a class's extrinsic features and adds them to an aspect.

```

class C {
    void foo(int a, int b){
        ...
    }
    void bar(int c) {
    }
}

class C {
    public void foo(int a,int b){
        ...
    }
    ...
}

aspect ExtrinsicC {
    public void C.bar(int c) {
        ...
    }
}

```

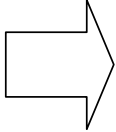


Fig. 6. Extracting method `bar` to aspect `ExtrinsicC`

In order to transform a method into an introduction⁶, we first have to check whether the selected destination aspect includes already existing field introductions with private access to the same target class. Then, have to check whether calls within the method body referring to identically named members could become ambiguous after the introduction. In AspectJ private introductions to a target class are accessible from other introductions within the aspect. Hence, this might cause a number of conflicts. Those ambiguous references cause no compilation errors, but yield to unintended behavior change. Because the introduction access modifier applies in relation to the aspect, not in relation to the target type, we have to check whether existing references referring to the focused method are valid any more. If one does not, we must consider weakening the access modifier of the introduction.

When all preconditions are fulfilled, we can introduce the method, since pointcuts are valid on method introductions as well. Hence, we do not have to check the enabling conditions.

⁶ In this section we concentrate on method introductions. However, field and parent introductions work likewise.

5.3 Separate Pointcut

If one starts to evolve a new aspect from scratch, it requires less effort and is at first more comfortable to implement the pointcuts as anonymous pointcut definitions. Anonymous pointcut means that someone defines a pointcut as right hand side part of a piece of advice without to declare an autonomous pointcut name. It is often observable that one applies such pointcuts with the intention to test first whether the newly defined pointcuts address the correct join points.

This approach is suitable as long as just a handful pieces of advice exist and the logical composed pointcut definitions do not evolve too large. When the aspect code evolves there are situations where pointcut definitions need to be reused in order to avoid numerous redundancies.

A first step is to transform the anonymous into a named pointcut. That facilitates the overview if large and complex pointcut compositions are further extended. Pointcut designators can be primitive or composite. Composite pointcut designators are composed primitive pointcut designators using the operators `|`, `&&`, and `!`.


<pre>pointcut x(C c, int a, int b): call(void C.foo(int, int)) && target(c) && args(a, b); pointcut y(C c, int a, int b): call(void C.bar(.., int)) && target(c) && args(a, b); after(C c, int a, int b) : x(c, a, b) y(c, a, b){...}</pre>		<pre>pointcut x(): call(void C.foo(int, int)); pointcut y(): call(void C.bar(.., int)); pointcut z(C c, int a, int b): target(c) && args(a, b); after(C c, int a, int b) : (x() y()) && z(c, a, b) {...}</pre>
--	--	--

Fig. 7. Separating pointcuts

The second step is to separate the composite pointcut into its single logical independent component pointcuts. Figure 7 illustrates partial redundant composite pointcut designators. These should better be separated in order to facilitate the composition and adaptability of pointcuts in the case of future refactorings or program adaptations. This is also an enabling approach to the redefinition of existing pointcuts in sub-aspects as described in [6].

Let us assume we have to compose the pointcut `call(void C.bar(.., int))` in a different context and want to negate it. The drawback would be that we have to define a new primitive pointcut designator in order to avoid the simultaneous negation of the two dynamic pointcuts in pointcut `y`. On the right side of figure 7 we present a more flexible implementation. After that transformation we can effortlessly build an expression like `!y()` and compose it with additionally named pointcuts. In this example we have just separated the context exposure and the signature matching parts to reduce redundancy and foster reusability. In other contexts it could be necessary to further separate the context exposing pointcut `z` as well.

6 Automated Aspect-Oriented Refactoring Eclipse

In this paper the term automated refactoring refers to tool supported *AspectJ* code-transformations and enabling condition checking. The essence of an aspect-aware refactoring tool is to provide the user with reliable information on enabling condition violations while performing various refactorings.

Already in small implementations, it is an exhausting task to manually verify only the first and the second enabling condition after each transformation. In larger applications is the manual verification of the third condition not feasible.

An aspect-aware refactoring tool has to automate these tasks and provide the developer with a kind of refactoring workflow. Due to the possible complexity of composed pointcut definitions, the user should still be integrated in this workflow. A tool is meant to visit all enabling condition violating pointcuts, making proposals and ask the user for a confirmation. Tool supported transformations should create new code and modify existing as long as the application provides the semantically same behaviour.

We developed an *Eclipse* plug-in tool that supports AspectJ aware refactorings in that IDE in order to achieve most of these requirements. This plug-in is based on AspectJ version 1.0.6. It provides the developer with the claimed refactoring workflow. The tool consists of three collaborating parts. We implemented coding wizards, which make the transformation operations transparent and offer graphical user interfaces used to configure and provide parameters for these transformations. Secondly, the tool possesses as a key component a transformation and code generation functionality which is based on the abstract syntax tree provided by the AspectJ compiler. The third part is an enabling condition checking functionality that we evolved through utilization of further features of the AspectJ compiler.

Similarly, to the large OO-Refactorings, this refactoring tool can automate the creation of parts of *AspectJ* idioms [4, 5]. The tool offers following refactorings: *Extract Advice*, *Extract Introduction*, *Introduce into Container* [4], *Separate Pointcut* [4], *Rename Method and Rename Pointcut*. Currently, our tool does not support a more sophisticated workflow as described above.

7 Conclusion

In this paper we discussed the interplay between object-oriented refactorings and aspect-oriented code. We showed that the application of object-oriented refactorings to the (object-oriented) base program of an aspect-oriented system is not behaviour preserving. The reason for it lies in the join point specification of aspects. As a consequence, current available refactoring tools cannot be applied to the base programs of aspect-oriented software. For development processes that highly depend on refactoring those aspect-oriented techniques cannot be applied. In detail, we discussed the fragility of join points specifications in AspectJ in conjunction with object-oriented refactorings.

We proposed AspectJ-aware modification of the well-known and often used refactorings *rename method*, *move method* and *extract method*. Those modifications permit

to transform the base program without changing the behaviour of the final woven AspectJ application. Furthermore, we proposed three aspect-oriented refactorings which are often needed either to migrate from object-oriented code to aspect-oriented code (*extract advice* and *extract introduction*) or to increase the comprehensibility of existing aspect-oriented code (*separate pointcut*). Afterwards we briefly introduced our refactoring browser for AspectJ applications.

The here proposed solution is closely related to AspectJ which highly makes use of lexical join point specifications. Since, AspectJ is currently used by a large community and plays already an important role in numerous software projects, the proposed refactorings and its tool support is of high practical interest. On the other hand, the proposed solution cannot be directly applied to other aspect-oriented techniques since different aspect-oriented techniques differ widely in their join point language. For example, join points in *Sally* [7] not necessarily depend that much on lexical join points. Hence, it is a future task to analyze different aspect-oriented techniques regarding their fragility of join point language in the context of refactoring.

Acknowledgement

Many thanks to Jan Wloka for his valuable comments on the paper.

References

1. AspectJ Team: *AspectJ Programming Guide*, available at: <http://www.eclipse.org/aspectj/>, 2001.
2. Beck, K.: *Extreme Programming Explained*, Addison-Wesley, 1999
3. Fowler, M.; Beck, K.; Brant, J.; Opdyke, W. F.; Roberts, D.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
4. Hanenberg, S.; Costanza, P.: *Connecting Aspects in AspectJ: Strategies vs. Patterns*, 1st Workshop on Aspects, Components and Patterns for Infrastructure Software, AOSD'01, Enschede, April, 2002.
5. Hanenberg, S.; Schmidmeier, A.: *Aspect-Oriented Idioms in AspectJ*, To appear in: proceedings of EUROPLOP 2003.
6. Hanenberg, S.; Unland, R.: *Using and Reusing Aspects in AspectJ*. Workshop on Advanced Separation of Concerns, OOSPLA, 2001.
7. Hanenberg, S.; Unland, R.: *Parametric Introductions*, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, MA, March 17 - 21, 2003, pp. 80-89.
8. Hirschfeld, R.: *Aspect-Oriented Programming with AspectS*, Proceedings of NetObject-Days 2002, Erfurt, pp. 219-235.
9. Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwing, J.: *Aspect-Oriented Programming*. Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag, 1997, pp. 220-242.
10. Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, J.: *An Overview of AspectJ*, Proceedings of European Conference on Object-Oriented Programming (ECOOP), LNCS 2072, Springer-Verlag, 2001, pp. 327-353.

11. Lieberherr, K.; Lorenz, D.; Mezini, M.: *Programming with Aspectual Components*, College of Computer Science, Northeastern University, Technical Report, NU-CCS-99-01, Boston, MA, 1999.
12. Masuhara, H.; Kiczales, G.; Dutchyn, C.: *A Compilation and Optimization Model for Aspect-Oriented Programs*, Proceedings of Compiler Construction (CC2003), LNCS 2622, pp.46-60, 2003
13. Popovici, A.; Gross, T.; Alonso, G.: *Dynamic Weaving for Aspect-Oriented Programming*, Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD), Enschede, The Netherlands, April 22-26, 2002, pp. 141 - 147.
14. Popovici, A.; Gross, T.; Alonso, G.: *Just in Time Aspects*, Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD), Boston, MA, March 17 - 21, 2003.
15. Opdyke, W. F.: *Refactoring Object-Oriented Frameworks*, Ph.D. thesis, University of Illinois, 1992.
16. Tokuda, L. A.: *Evolving Object-Oriented Designs with Refactorings*, Ph.D. thesis, University of Texas at Austin, 1999.