Refactoring Aspect-Oriented Programs

Masanori Iwamoto
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku
Fukuoka 811-0295, Japan
mfm03005@ws.ipc.fit.ac.jp

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku
Fukuoka 811-0295, Japan
zhao@cs.fit.ac.jp

1. INTRODUCTION
Refactoring is the process of changing a program to improve its internal structure and reusability, without altering the external behavior of the program. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In this paper, we propose a systemic approach to refactoring aspect-oriented programs. To this end, we first investigate the impact of existing aspect-oriented refactorings such as those proposed by Fowler [4] on aspect-oriented programs. Then we propose some new aspect-oriented refactorings that are unique to aspect-oriented programs. Finally, we discuss tool support for automatic refactoring of aspect-oriented programs. We use AspectJ, a general-purpose AOP language to demonstrate our approach, but our approach is general enough to be applicable to other AOP languages.

AOP is a programming technique for expressing programs involving encapsulated, crosscutting concerns through composition techniques, and through reuse of the crosscutting code [2, 6, 8, 11]. The AOP is able to modularize crosscutting aspects of a software system. As objects in object-oriented software, aspects in aspect-oriented software may arise at any stage of the software life cycle, including requirements specification, design, implementation, etc. Some examples of crosscutting aspects are exception handling, synchronization, and resource sharing.

The current research so far in AOP is focused on problem analysis, software design, and implementation techniques. Even if refactoring is important to improve software quality, development of refactorings and their tool support for aspect-oriented software is still ignored during the current stage of the technical development. Since aspect-oriented programming introduces some new kinds of modules such as advice, introduction, pointcuts, and aspects that are different from methods in a class, existing approaches to refactoring procedural and object-oriented programs can not be directly applied to the AOP domain. In order to improve the code quality for aspect-oriented software, refactorings and their tool support that are appropriate for aspect-oriented programs are required.

However, although refactoring has been studied widely for procedural and object-oriented software, the development of refactoring patterns for aspect-oriented software is just starting. We know that several researchers [3, 5, 9, 12] are studying this problem recently, but neither of them has demonstrated detail information on how to refactoring of aspect-oriented software.

When performing refactoring on aspect-oriented programs, we are interested in the following questions.

1. Can existing object-oriented refactorings such as those proposed by Fowler [4] be adopted to aspect-oriented programs? If not, how should we do when applying these refactorings to aspect-oriented programs?

2. Are there some new refactorings that are unique to aspect-oriented programs but different from existing object-oriented refactorings?

3. How to support automatic refactoring of aspect-oriented programs?

In this paper, we propose a systemic approach to refactoring aspect-oriented programs. We study this problem from three aspects. First, we investigate whether the object-oriented refactorings such as those proposed by Fowler [4] can be applied to aspect-oriented code; we propose some modification guidelines in order to adopt these refactorings to the
domain of AOP. Second, by carefully studying the concept and structure of aspect-oriented languages, we identify some new refactorings that are unique to aspect-oriented code but different from existing object-oriented refactorings. Finally, we discuss issues on automatic support for refactoring of aspect-oriented programs based on control flow and data flow analysis.

Because AOP is a new language paradigm that is different from procedural and object-oriented language, we really need to develop a systematic approach to supporting refactoring of aspect-oriented software. We hope that by examining the ideas of aspect-oriented refactoring from several different viewpoints and through independently developed aspect-oriented refactorings, we can have a better understanding of what the refactoring is meant in the AOP domain and the role that refactorings plays in the development of quality aspect-oriented software. As the first step, this paper is to report our primary results on refactoring of aspect-oriented software.

The rest of the paper is organized as follows. Section 2 briefly introduces AspectJ, a general aspect-oriented programming language based on Java. Section 3 discusses how existing object-oriented refactorings can be applied to aspect-oriented programs. Section 4 proposes some new refactorings that are unique to aspect-oriented programming. Section 5 discusses tool support for refactoring of aspect-oriented programs. Section 6 discusses some related work, and concluding remarks are given in Section 7.

2. ASPECT-ORIENTED PROGRAMMING WITH ASPECTJ

We present our refactoring approach for aspect-oriented programs in the context of AspectJ, the most widely used aspect-oriented programming language [1, 7]. Our basic techniques, however, deal with the basic concepts of aspect-oriented programming and therefore apply to the general class of aspect-oriented languages.

AspectJ [1] is a seamless aspect-oriented extension to Java. AspectJ adds to Java some new concepts and associated constructs. These concepts and associated constructs are called join point, pointcut, advice, introduction, and aspect.

Aspect is a modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that crosscuts other classes in a program. An aspect is defined by aspect declaration, which has a similar form of class declaration in Java. Similar to a class, an aspect can be instantiated and can contain state and methods, and also may be specialized in its sub-aspects. An aspect is then combined with the classes it crosscuts according to specifications given within the aspect. Moreover, an aspect can introduce methods, attributes, and interface implementation declarations into types by using the introduction construct. Introduced members may be made visible to all classes and aspects (public introduction) or only within the aspect (private introduction), allowing one to avoid name conflicts with pre-existing members.

The essential mechanism provided for composing an aspect
/* Before refactoring */
aspect AspectSample {
    before(): call(* Sample.pm()) {
        System.out.println("pm ok");
    }
}
class Sample {
    public static void main(String args[]) {
        new Sample().pm();
    }
    void pm() {
        System.out.println("print method");
    }
}

/* After refactoring */
aspect AspectSample {
    before(): call(* Sample.pm()) {
        System.out.println("pm ok");
    }
}
class Sample {
    public static void main(String args[]) {
        new Sample().print_method();
    }
    void print_method() {
        System.out.println("print method");
    }
}

Figure 2: An OO-refactoring for renaming method.

The program contains one aspect PointShadowProtocol and two classes Point and Shadow. The aspect has three methods: getShadowCount, associate and getShadow, and three pieces of advice related to pointcuts setting, settingX and settingY respectively. The aspect also has two attributes shadowCount and shadow such that shadowCount is an attribute of the aspect itself and shadow is an attribute that is privately introduced to class Point.

3. OBJECT-ORIENTED REFACTORINGS
We study how existing object-oriented refactorings can be applied to aspect-oriented programs.

3.1 Motivating Examples
We present two examples to explain the problems when applying existing object-oriented refactoring (OO-refactoring for short) to aspect-oriented programs.

Figure 2 presents a program containing a class Sample in which a main method and a pm method are declared, and an aspect AspectSample in which a piece of before advice is declared. The advice can be applied to each join point where a target object of type Sample receives a call to its method with signature call(* Sample.pm()).

Example. Figure 1 shows an AspectJ program taken from [1] that associates shadow points with every Point object.

/* Before refactoring */
aspect AspectSample {
    before(): call(void Sample.setA(int)) {
        System.out.println("method ok");
    }
}
class Sample {
    private int a;
    public static void main(String args[]) {
        new Sample().print_method(10);
    }
    void setA(int x) {
        a = x;
    }
    void print_method(int x) {
        setA(x);
        System.out.println("setA");
    }
}

/* After refactoring */
aspect AspectSample {
    before(): call(void Sample.setA(int)) {
        System.out.println("method ok");
    }
}
class Sample {
    private int a;
    public static void main(String args[]) {
        new Sample().print_method(10);
    }
    void print_method(int x) {
        a = x;
        System.out.println("setA");
    }
}

Figure 3: An OO-refactoring for removing method.

Unlike a method that has a unique name, advice in AspectJ has no name. So for easy expression, we use the name of a pointcut to stand for the name of advice it is associated with.


<table>
<thead>
<tr>
<th>OO Refactoring</th>
<th>Impact by Aspects</th>
<th>OO Refactoring</th>
<th>Impact by Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Parameter</td>
<td>○</td>
<td>Extract Interface</td>
<td>○</td>
</tr>
<tr>
<td>Extract Class</td>
<td>○</td>
<td>Extract Superclass</td>
<td>○</td>
</tr>
<tr>
<td>Extract Method</td>
<td>○</td>
<td>Inline Class</td>
<td>○</td>
</tr>
<tr>
<td>Extract Subclass</td>
<td>○</td>
<td>Inline Method</td>
<td>○</td>
</tr>
<tr>
<td>Hide Method</td>
<td>○</td>
<td>Introduce Explaining Variable</td>
<td>○</td>
</tr>
<tr>
<td>Inline Method</td>
<td>○</td>
<td>Move Field</td>
<td>○</td>
</tr>
<tr>
<td>Parameterize Method</td>
<td>○</td>
<td>Move Setting Method</td>
<td>○</td>
</tr>
<tr>
<td>Pull Up Field</td>
<td>○</td>
<td>Replace Array with Object</td>
<td>○</td>
</tr>
<tr>
<td>Push Down Field</td>
<td>○</td>
<td>Replace Conditional with Polymorphism</td>
<td>○</td>
</tr>
<tr>
<td>Rename Method</td>
<td>○</td>
<td>Replace Magic Number with Symbolic Constant</td>
<td>△</td>
</tr>
<tr>
<td>Replace Parameter with Explicit Methods</td>
<td>○</td>
<td>Replace Nested Conditional with Guard Clauses</td>
<td>△</td>
</tr>
<tr>
<td>Remove Parameter</td>
<td>○</td>
<td>Sel Encapsulate Field</td>
<td>○</td>
</tr>
</tbody>
</table>

○: Modification is needed when applying to AOP  △: Applicable to AOP directly without modification

3.2 Discussions

Most of object-oriented refactorings, as we showed above, are not valid when applying to AspectJ code. They have to be adapted to consider the impact that the modifications on

toring on method \texttt{pm} to rename its name from \texttt{pm} to \texttt{print\_method}. We can simply change all the places that \texttt{pm} is occurred in class \texttt{Sample} using an editor, or a refactoring tool. After that, however, we found that the program's behavior has also been changed, which is not the case we would like to. By carefully examining the source code. We found the problem: since the before advice relies on the method call join point that is related to method signature \texttt{pm}, in addition to the above operations, we should also modify the pointcut to make it point to method \texttt{print\_method}. Otherwise, the program may produce an unexpected result.

Similar problems may occur in other refactoring cases. Figure 3 presents another program which contains a class \texttt{Sample} and an aspect \texttt{AspectSample}. \texttt{Class Sample} declares three methods: \texttt{main}, \texttt{setA}, and \texttt{print\_method}. \texttt{AspectSample} declares a piece of before advice. The advice can be applied to each join point where a target object of type \texttt{Sample} receives a a call to its method with signature \texttt{call void \texttt{Sample.setA} (int)}. By examining the code, we found that the code quality can be improved by a removing-method refactoring operating on methods \texttt{setA} and \texttt{print\_method}. To do so, we put \texttt{setA}’s content into \texttt{print\_method} and remove \texttt{setA} entirely. After that, however, we found that when ran the program we got an unexpected result. The reason for this problem is that we ignored the impact from the aspect \texttt{AspectSample} during the refactoring, which leads to a similar problem as we presented in the previous example.

Since existing object-oriented refactorings operate only on classes (objects), they cannot solve these problems demonstrated above when applied to aspect-oriented programs. It is therefore not reasonable to simply apply existing object-oriented refactorings to aspect-oriented programs without considering the impact from aspects. In order to use object-oriented refactoring for aspect-oriented programs, modifications for these refactorings and some guidelines are needed. In Section 3, we will discuss this issue in greater detail.

Figure 4: An AO-refactoring for extracting pointcut.
4. ASPECT-ORIENTED REFACTORINGS

We next present some new aspect-oriented refactorings that are unique to aspect-oriented programs, but different from existing object-oriented refactorings.

4.1 Motivating Examples

We present two examples to explain the problems when perform aspect-oriented refactoring (AO-refactoring for short).

Figure 4 presents a program containing an aspect \texttt{AspectSample} in which a piece of before advice and a piece of after advice are declared, and a class \texttt{Class1} in which a main method and a method are declared. Both the before and after advice can be applied to the same join point where a target object of type \texttt{Sample} receives a call to its method with signature call(* \texttt{Sample.pm}()). In order to reuse this join point we may perform a refactoring to extract the pointcut attached to both the before and after advice to form a new pointcut \texttt{methodCall}. By doing so, the extracted pointcut \texttt{methodCall} can be reused by other advice as well. We call such a refactoring as \textit{Extract Pointcut}.

Figure 5 presents another program containing a method \texttt{rent} which has a precondition declared by if statement. The code is taken from [9]. In order to reuse the precondition, we may turn it into a piece of before advice whose name reflects the value that the advice gives. We also need a pointcut check to represent the join point related to the advice. This refactoring operation is called \textit{Extract Advice} [9].

Refactorings such as \textit{Extract Pointcut} and \textit{Extract Advice} showed above are generally different from existing object-oriented refactorings, and we call them \textit{aspect-oriented refactors}. The difference between object-oriented and aspect-oriented refactorings is that the former focuses only on objects (classes) whereas the later has to focus on both objects and aspects.

4.2 A Catalog of Aspect-Oriented Refactorings

Aspect-orientation introduces new aspect-aware refactorings that differ from existing object-oriented refactorings. These aspect-oriented refactorings should be identified, and a catalog for these refactorings should be presented.

Table 1 lists aspect-oriented refactorings we proposed; we denote these refactorings as \textit{AO-refactorings} for simplifying explanation. Note that this is just a primary list of aspect-oriented refactorings we identified, and more AO-refactorings will be added to the list as we get some new results.

Due to space limitation, however, we can not explain these AO-refactorings listed in Table 2: one can refer to [13] to obtain greater detail information for these AO-refactorings.

5. TOOL SUPPORT

Tool support is essential for any refactoring techniques. Without tool support, refactorings can not be applied to large-scale systems, and therefore lose practices. In this section, we describe our refactoring tool called \texttt{AspectJ Refactoring Tool} (ART for short), that supports refactoring of AspectJ programs. Unlike most existing refactoring tools for object-oriented code that mainly operate on the abstract syntax trees (ASTs), ART uses the \textit{program dependence graph} (PDG) as a basic abstract data structure for representing aspect-oriented programs. ART therefore op-
erates on the PDG to perform refactoring on the programs. The reason for using the PDG is that we can automatically realize more refactoring patterns than with those AST-based tools.

Figure 6 shows the basic structure of ART which mainly consists of two components, i.e., \textit{refactoring component} and \textit{user interface component}. The refactoring component is further consists of a parser for AspectJ code, a dependence analyzer, a code transformer, a refactoring operator, and an operation sequence indexing part.

6. RELATED WORK

We discuss related work in the area of refactoring for object-oriented and aspect-oriented programs.

During the last decade, refactoring object-oriented programs has become a very active research area in software engineering community, and many refactorings and their support tools for object-oriented programs have been developed [4, 10]. However, as we discussed in Sections 3 and 3, most of these refactorings can not be directly applied to aspect-oriented programs because they can not handle the impact problem arose from refactoring of aspect-oriented software.

Several research groups are studying the problem of refactoring aspect-oriented software with different approaches and from different viewpoints. Włoda [12] explored the relationship between refactoring and aspect-orientation, but did not directly address the issue on how to perform refactorings on aspect-oriented programs. Borba and Soares [3] proposed a program transformation based approach to developing refactoring and code generation tools for AspectJ, but did not provide details on their approach. Hannemann [5] is working on dialogue-based aspect-oriented refactoring which focuses on study aspect-oriented design pattern refactorings. However, no detail about his approach is available now. Miller [9] proposed two aspect-oriented refactorings called \textit{Extract Advice} and \textit{Extract Introduction} which is a very small subset of our catalog for aspect-oriented refactorings.

7. CONCLUDING REMARKS

In this paper we proposed a systemic approach to refactoring aspect-oriented software. To this end, we first investigated the impact of existing object-oriented refactorings such as those proposed by Fowler [4] on aspect-oriented programs and gave some guidelines for solving these problems. We then proposed some new aspect-oriented refactorings that are unique to aspect-oriented programs but different from existing aspect-oriented refactorings. Finally, we discussed some implementation issues on our tool for supporting automatic refactoring of AspectJ programs.

8. REFERENCES

Figure 6: The structure of AspectJ Refactoring Tool (ART).

