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

ABSTRACT

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 object-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.

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. Essentially, when developers refactor they are improving the design of code while it is being written, and in fact, during the entire software life cycle. Refactoring tools helps software developers to safely and efficiently restructure their code to greatly improve its quality, reliability, and maintainability.

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

```
 ce0 public class Point {
  s1   protected int x, y;
 me2   public Point(int _x, int _y) {
  s3     x = _x;
  s4     y = _y;
       }
 me5   public int getX() {
  s6     return x;
       }
 me7   public int getY() {
  s8     return y;
       }
 me9   public void setX(int _x) {
 s10     x = _x;
       }
me11   public void setY(int _y) {
 s12     y = _y;
       }
me13   public void printPosition() {
 s14     System.out.println("Point at("+x+","+y+")");
       }
me15   public static void main(String[] args) {
 s16     Point p = new Point(1,1);
 s17     p.setX(2);
 s18     p.setY(2);
       }
     }
ce19 class Shadow {
 s20   public static final int offset = 10;
 s21   public int x, y;

me22   Shadow(int x, int y) {
 s23     this.x = x;
 s24     this.y = y;
me25   public void printPosition() {
 s26     System.outprintln("Shadow at
                ("+x+","+y+")");
       }
     }
```

```
ase27 aspect PointShadowProtocol {
  s28   private int shadowCount = 0;
 me29   public static int getShadowCount() {
  s30     return PointShadowProtocol.
                aspectOf().shadowCount;
       }
  s31   private Shadow Point.shadow;
 me32   public static void associate(Point p, Shadow s){
  s33     p.shadow = s;
       }
 me34   public static Shadow getShadow(Point p) {
  s35     return p.shadow;
       }

 pe36   pointcut setting(int x, int y, Point p):
          args(x,y) && call(Point.new(int,int));
 pe37   pointcut settingX(Point p):
          target(p) && call(void Point.setX(int));
 pe38   pointcut settingY(Point p):
          target(p) && call(void Point.setY(int));

 ae39   after(int x, int y, Point p) returning :
            setting(x, y, p) {
  s40     Shadow s = new Shadow(x,y);
  s41     associate(p,s);
  s42     shadowCount++;
       }
 ae43   after(Point p): settingX(p) {
  s44     Shadow s = new getShadow(p);
  s45     s.x = p.getX() + Shadow.offset;
  s46     p.printPosition();
  s47     s.printPosition();
       }
 ae48   after(Point p): settingY(p) {
  s49     Shadow s = new getShadow(p);
  s50     s.y = p.getY() + Shadow.offset;
  s51     p.printPosition();
  s52     s.printPosition();
     }
   }
```

Figure 1: A sample AspectJ program.

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 systemic 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.

with other classes is called a *join point*. A join point is a well-defined point in the execution of a program, such as a call to a method, an access to an attribute, an object initialization, exception handler, etc. Sets of join points may be represented by *pointcuts*, implying that such sets may crosscut the system. Pointcuts can be composed and new pointcut designators can be defined according to these combinations. AspectJ provides various pointcut *designators* that may be combined through logical operators to build up complete descriptions of pointcuts of interest. For a complete listing of possible designators one can refer to [1].

An aspect can specify *advice* that is used to define some code that should be executed when a pointcut is reached. Advice is a method-like mechanism which consists of code that is executed *before*, *after*, or *around* a pointcut. `around` advice executes *in place* of the indicated pointcut, allowing a method to be replaced.

An AspectJ program can be divided into two parts: *base code* part which includes classes, interfaces, and other language constructs for implementing the basic functionality of the program, and *aspect code* part which includes aspects for modeling crosscutting concerns in the program. Moreover, any implementation of AspectJ should ensure that the base and aspect code run together in a properly coordinated fashion. Such a process is called *aspect weaving* and involves making sure that applicable advice runs at the appropriate join points. For detailed information about AspectJ, one can refer to [1].

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.

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[1]. 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())`.

Suppose we would like to perform an object-oriented refac-

[1]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 associated with.

Table 1: Impact on applying OO refactorings to AOP

OO Refactorings	Impact by Aspects	OO Refactorings	Impact by Aspects
Add Parameter	◯	Encapsulate Downcast	◯
Extract Class	◯		
Extract Method	◯	Extract Interface	◯
Extract Subclass	◯	Extract Superclass	◯
Hide Method	◯	Inline Class	◯
Inline Method	◯	Inline Temp	△
Introduce Explaining Variable	◯	Move Field	◯
Move Method	◯	Move Setting Method	◯
Parameterize Method	◯	Pull Up Constructor	◯
Pull Up field	◯	Pull Up Method	◯
Push Down Field	◯	Push Down Method	◯
Rename Method	◯	Replace Array with Object	◯
Replace Conditional with Polymorphism	◯	Replace Exception with Test	◯
Replace Magic Number with Symbolic Constant	△	Replace Nested Conditional with Guard Clauses	△
Replace Parameter with Explicit Methods	◯	Replace Temp with Query	◯
Remove Parameter	◯	Self Encapsulate Field	◯

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

toring on method `pm` to rename its name from `pm` to `print_method`. We can simply change all the places that `pm` is occurred in class `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 `pm`, in addition to the above operations, we should also modify the pointcut to make it point to method `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 `Sample` and an aspect `AspectSample`. Class `Sample` declares three methods: `main`, `SetA`, and `print_method`. Aspect `AspectSample` declares a piece of before advice. 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(void Sample.setA(int))`. By examining the code, we found that the code quality can be improved by a removing-method refactoring operating on methods `setA` and `print_method`. To do so, we put `setA`'s content into `print_method` and remove `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 `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 can not 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.

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

```
/* Before refactoring */
class Class1 {
    public static void main(String args[]) {
        Class1 c1 = new Class1();
        c1.method();
    }
    void void method() {
        System.out.println("method");
    }
}
aspect AspectSample {
    before(): call(void Class1.method()) {
        System.out.println("before method");
    }
    after(): call(void Class1.method()) {
        System.out.println("after method");
    }
}

/* After refactoring */
class Class1 {
    public static void main(String args[]) {
        Class1 c1 = new Class1();
        c1.method();
    }
    void void method() {
        System.out.println("method");
    }
}
aspect AspectSample {
    pointcut methodCall(): call(void Class1.method());
    before(): methodCall() {
        System.out.println("before method");
    }
    after(): methodCall() {
        System.out.println("after method");
    }
}
```

Figure 4: An AO-refactoring for extracting pointcut.

```
/* Before refactoring */
public void rent(Customer customer) throws
                    IllegalArgumentException {
    if (customer == null) {
          throw new IllegalArgumentException
                          (``The argument is null'');
    }
    customer.addVideo(this);
}

/* After refactoring */
public void rent(Customer customer) throws
                      IllegalArgumentException {

    customer.addVideo(this);
}

public aspect NullArgumentChecker {
    pointcut check(Customer customer):
         call(void Copy.rent(Customer customer));
    before(): check(Customer customer) {
        if (customer == null)
            throw new IllegalArgumentException
                            ("The argument is null");
    }
}
```

Figure 5: An AO-refactoring for extracting advice.

the base code (i.e., Java code) may have effects on the corresponding aspect code. Therefore, any refactorings being applied to aspect-oriented code must consider such effect.

Here we discuss some issues on applying existing object-oriented refactorings proposed by Fowler [4] to AspectJ code and give some guidelines for how to avoid invalid programs when adapting these refactorings to AspectJ code. Table 1 lists the refactorings we adopted from Fowler's book [4]; we denote these refactorings as *OO-refactorings* for simplifying explanation. In the table, the first column lists OO-refactorings we chose for investigation; the second column shows whether the impact from aspects should be considered or not.

We found that almost all OO-refactorings we investigated have the similar effect problem when applied directly to aspect code; only several refactorings seem no problem and therefore applicable directly to AspectJ code without any modification. These refactorings include *Consolidate Duplicate Conditional Fragments*, *Inline Temp*, *Remove Assignments to Parameters*, *Replace Nested Conditional with Guard Clauses*, *Replace Magic Number with Symbolic Constant*, and *Split Temporary Variable*.

Due to space limitation, however, we can not report our results for all these OO-refactorings listed in Table 1; one can refer to [13] to obtain greater detail information for these OO-refactorings.

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 `AspectSample` in which a piece of before advice and a piece of after advice are declared, and a class `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 `Sample` receives a call to its method with signature `call(* 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 `methodCall`. By doing so, the extracted pointcut `methodCall` can be reused by other advice as well. We call such a refactoring as *Extract Pointcut*.

Figure 5 presents another program containing a method `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 *Extract Advice* [9].

Refactorings such as *Extract Pointcut* and *Extract Advice* showed above are generally different from existing object-oriented refactorings, and we call them *aspect-oriented refactorings*. 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 *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 *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 *program dependence graph* (PDG) as a basic abstract data structure for representing aspect-oriented programs. ART therefore op-

Table 2: A Catalog of aspect-oriented refactorings

AO Refactorings	AO Refactorings
Extract Advice	Extract Introduction
Extract Pointcut	Inline Pointcut
Remove This Pointcut	Remove Target Pointcut
Reference Pointcut	Introduce Declare Parents
Introduce Abstract Aspect	Pull Up Pertarget
Move to Advice	Move to Introduction
Move Class Method to Aspect	Move Aspect Method to Class
Move Class Field to Aspect	Transform Class Introduction into Aspect Method
Transform Aspect Method into Class Introduction	Transform Class Introduction into Aspect Field
Transform Aspect Field into Class Introduction	Introduce Execution Pointcut
Introduce Call Pointcut	Introduce Around Advice
Combine Pointcut	Decompose Pointcut

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., *refactoring component* and *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. Wloka [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 *Extract Advice* and *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 object-oriented refactorings. Finally, we discussed some implementation issues on our tool for supporting automatic refactoring of AspectJ programs.

8. REFERENCES

[1] The AspectJ Team. The AspectJ Programming Guide. 2002.

[2] L. Bergmans and M. Aksits. Composing crosscutting Concerns Using Composition Filters. *Communications of the ACM*, Vol.44, No.10, pp.51-57, October 2001.

[3] P. Borba and S. Soares. Refactoring and Code Generation Tools for AspectJ. October 2002.

[4] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman, 1999.

[5] J. Hannemann. Dialogue-Based Aspect-Oriented Refactoring. http://www.cs.ubc.ca/labs/spl/projects/ao-refactoring.html

[6] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *Proceedings of the 11th European Conference on Object-Oriented Programming*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "An Overview of AspectJ," *proc. 13th European Conference on Object-Oriented Programming*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 2000.

[8] K. Lieberher, D. Orleans, and J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, Vol.44, No.10, pp.39-41, October 2001.

[9] G. Miller. Refactoring with Aspects. *Proc. 4th International Conference on Extreme Programming*, Genova, Italy, May 2003.

[10] W. F. Opdyke. Refactoring Object-Oriented Frameworks. Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1992.

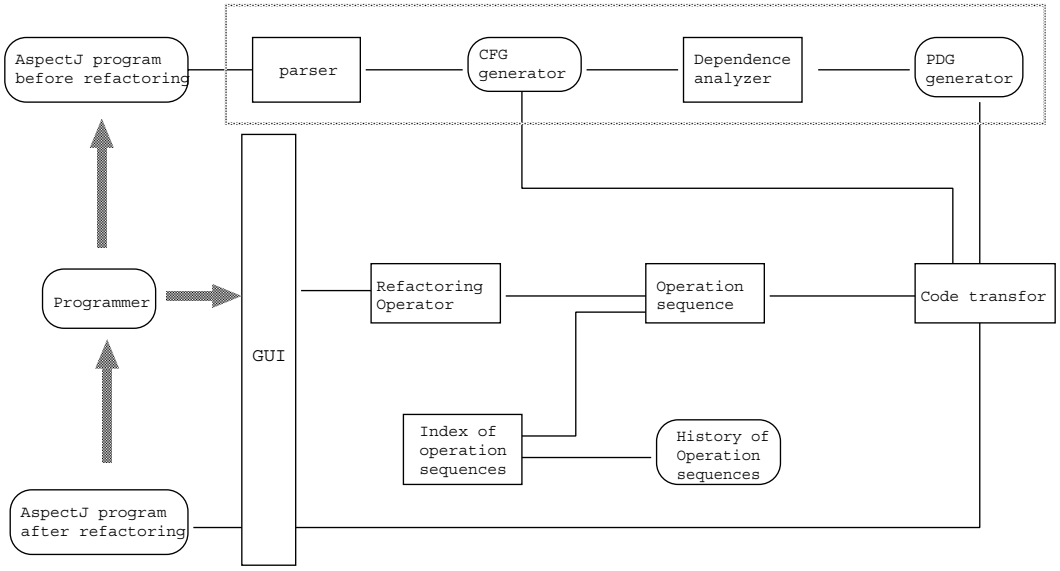

Figure 6: The structure of AspectJ Refactoring Tool (ART).

[11] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. *Proceedings of the International Conference on Software Engineering*, pp.107-119, 1999.

[12] J. Wloka. Refactoring in the Presence of Aspects. *ECOOP2003 PhD workshop*, July 2003.

[13] M. Iwamoto and J. Zhao. A Systemic Approach to Refactoring Aspect-oriented Programs. August 2003. (In Preparation)