Some thoughts on Refactoring for Aspect Oriented Programming using AspectJ

Geeta Bagade(Mete), Dr. Shashank Joshi

Abstract— Aspect Oriented programming (AOP) is not something new. But it has caught the attention of the developers recently. AOP's main aim to provide better means to address the issue of separation of concerns.AOP is implemented by using a variety of tools. These tools are an extension to the programming languages that already exist. AspectJ is one such language which is an extension to the Java Programming Language. Refactoring the existing system requires us to change the code . So, refactoring is a process of changing the software system in such a way that the behavior of the program does not change. In this paper, we propose a new set of refactoring that can be applied to Aspect Oriented Programs.

Index Terms—Refactoring, Aspect Oriented Programming, AOP, Aspect Oriented Systems, Concerns, AspectJ, Pointcut, Joinpoint

1 INTRODUCTION

Aspect Oriented Programming (AOP) :It is a paradigm that supports two main goals

1. Separation of Concerns
2. A mechanism to describe the concerns that cross-cut other components in the system.

It is implemented by using a wide set of tools that are specific to that programming language. These tools are extensions to the programming languages that already exist. AspectJ is one such language which is an extension to Java programming language. The language constructs of AspectJ include

1. Aspect: It is similar to a class. It defines the pointcut and the advice. It is compiled by using the AspectJ compiler which weaves the concerns into the objects that already exist.
2. Joinpoint: It is a point of execution in the program
3. Pointcut: It is the place where advices can be inserted
4. Advice: It is a construct that tells which code should execute at the join point. The code can execute before, after or around a join point. A "before" advice will run before the code at the joinpoint. The "after" advice will execute after the code at the join point. The "around" advice surrounds the code that exists at the joinpoint.

Refactoring: It is a process of changing the existing system or software in such a way that the behaviour of the program or the system does not change. Refactoring can be done manually as well as by using some refactoring tools

- *Geeta Bagade(Mete) is currently pursuing her Ph.D in Computer Science from Bharati Vidyapeeth Deemed University, Pune,India E-mail: geetamete@gmail.com*
- *Dr.Shashank Joshi is the Ph.D Guide. He works as a professor in BharatiVidyapeeth Deemed University's Engineering College,Pune, India. E-mail: sdj@live.in*

1.1 Refactoring Techniques:

There are a number of techniques used for refactoring. Some of them are

1. Assertions
2. Graph Transformations
3. Program Slicing
4. Software Metrics
5. Formal Concept Analysis
6. Program Refinement

Assertion technique can be used to express properties that should hold before the refactoring is applied and after the refactoring is applied. In Graph Transformations, every refactoring corresponds to a graph production rule. Each refactoring application also corresponds to a graph transformation. Program Slicing deals with restructuring function or procedure extraction. This technique can be used to guarantee that the refactoring will preserve the behaviour of interest. Software metrics is used before refactoring is applied to measure the quality of the software and then identify the places in the software that need refactoring. It is then used after the refactoring is done to measure the improvements that have taken place in the software. Formal Concept analysis can be used to restructure object oriented class hierarchies in a way that the behaviour is preserved. The program refinement technique is used to express the changes in the program in a formal way such that the behaviour is preserved. Program refinement technique is used in the experiments which will be covered in the later part of this paper

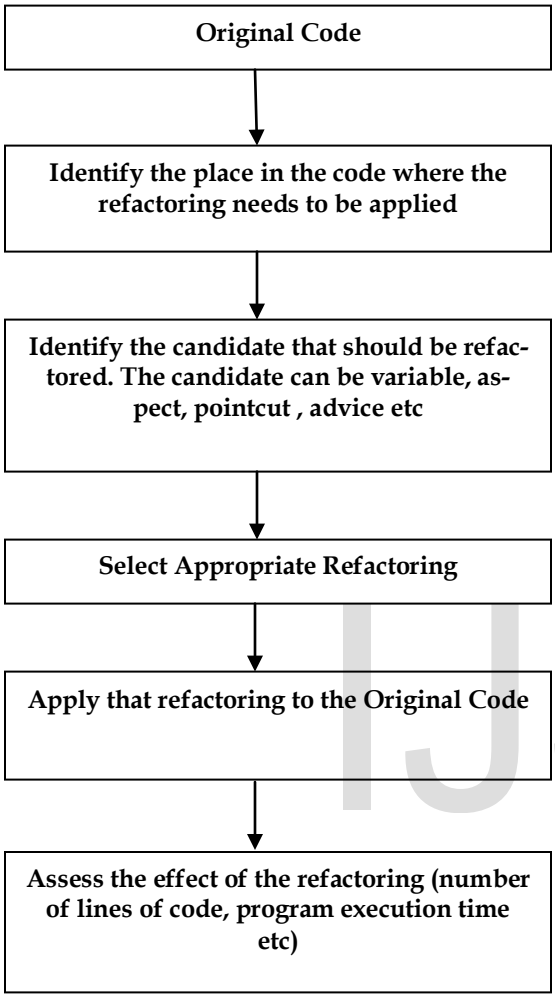
1.2 The Refactoring Process:

This process consists of six steps as mentioned below

1. Identify the place in the code where the software needs to be refactored.
2. Find out which refactoring should be applied in the code
3. Guarantee that the refactoring preserves the behaviour of the software
4. Apply the refactoring
5. Assess the effect of the refactoring
6. Maintain the consistency between the refactored code and the software artifacts like documentation, design documents, requirement specifications, tests etc.

Although there are plenty of refactorings available in this domain of AOP, there is a need to investigate other refactoring that will help is maintaining the software along with preserving its behaviour

1.3 The Refactoring Process Model



2 New Refactorings Identified:

2.1 Name of the refactoring: Make the aspect unprivileged: There are some aspects in which the advice or inter-type members need to access the private or protected resources of other types. To allow this aspects are declared as privileged. A privileged aspect can access the private inter-type declarations made by other aspects. So code in privileged aspect can access all its members even those that are private.
The proposed refactoring is to remove the keyword "privileged" from the aspect and make it unprivileged so that it cannot access the private members.

2.1.1 Refactoring Mechanics

1. Introduce a public member function that accesses the private data or method
2. In the aspect code, wherever there are references made to the variable that is private , replace it with objectname.functionname
3. Test if the restructured code preserves the behaviour

2.1.2 Experimented Code that contains the aspect that is privileged

```
privileged aspect A {
static final int MAX = 1000;
before(int x, C c): call(void C.incI(int)) && target(c) && args(x)
{
            if (c.i+x > MAX) throw new RuntimeException();
    }
 }


class C {
private int i = 0;
void incI(int x) { i = i+x; }
public static void main(String[] args) {
        C c = new C();
        c.incI(10);
        System.out.println("Working Prototype");
        }

}
```

In the above code there is a variable "i" which is declared as private. Since the aspect is declared as privileged, it is able to directly access that variable. Now let us remove the keyword "privileged" and refactor the code.

2.1.3 Code after applying the refactoring

```
aspect A {
        static final int MAX = 1000;
        before(int x, C c): call(void C.incI(int)) && target(c)
        && args(x) {
                if (c.getI()+x > MAX)
                        throw new RuntimeException();
            }
 }

class C {
        private int i = 0;
        void incI(int x) { i = i+x;
        }

        int getI()
        {
                return i;
        }

        public static void main(String[] args) {
                C c = new C();
                c.incI(10);
                System.out.println("Working Prototype");
        }
```

```
}
```

As seen in the refactored code above, the place where the variable i was accessed is replaced with the method call. This method is declared inside the class and is used in the aspect.

2.2 Name of the refactoring: Replace the pointcut name with its designator

A pointcut is a construct that tells the AOP language when it should match the join point. The purpose of a pointcut is to group the designators. A pointcut designator identifies the pointcut either by its name or by an expression. A pointcut can be declared inside an aspect, a class or an interface. Most of the pointcuts have a specific syntax as shown here

[access specifier] pointcut pointcutName(arguments): poincut-definition.

The access specifier can be public, private etc. Proposed refactoring is as described below. We have a pointcut written as shown below

```
pointcut P1() : call (int getI());
```

The advice written is

```
before(int x, C c): P1() && target(c) && args(x)
```

After refactoring the code , the statement P1() should be replaced by its actual definition as shown below

```
before(int x, C c): call(void C.incI(int)) && target(c) && args(x)
```

2.2.1 Refactoring Mechanics

1. Identify the pointcut that should be refactored
2. Replace the name of the pointcut with its designators
3. Test if the restructured code preserves the behaviour

2.2.2 Experimented Code that contains the pointcut name

```
aspect A {
        static final int MAX = 1000;
        pointcut P1() : call (int getI());
        pointcut P2() : call (void incI(int ));
        pointcut P3() : execution (int getI());
        pointcut P4() : execution (void incI(int )) ;

        before(int x, C c): P2() && target(c) && args(x) {
          System.out.println("Before Calling IncI");
           System.out.println(thisJoinPoint.getSignature());
           if (c.getI()+x > MAX)
                throw new RuntimeException();          }
        before() : P1() {
          System.out.println("Before calling getI");
          System.out.println(thisJoinPoint.getSignature());
}

after() : P1() {
  System.out.println("After calling getI");
  System.out.println(thisJoinPoint.getSignature());
}

before() : P3() {
  System.out.println("Before executing getI");
  System.out.println(thisJoinPoint.getSignature());
}

after() : P3() {
  System.out.println("After executing getI");
  System.out.println(thisJoinPoint.getSignature());
}

after(int x, C c): P2() && target(c) && args(x) {
  System.out.println("After Calling IncI");
  System.out.println(thisJoinPoint.getSignature());
}

before(int x, C c): P4() && target(c) && args(x){
  System.out.println("Before Executing IncI");
  System.out.println(thisJoinPoint.getSignature());
}

after(int x, C c): P4() && target(c) && args(x) {
  System.out.println("After Executing IncI");
  System.out.println(thisJoinPoint.getSignature());
    }

class C {
        private int i = 0;
        void incI(int x) { i = i+x; }
        int getI(){
                return i;
           }
        public static void main(String[] args) {
                C c = new C();
                c.incI(10);
                System.out.println("Working Prototype");
                }

}
```

2.2.3 Refactored code that contains the pointcut designator

```
aspect A {
        static final int MAX = 1000;
        before(int x, C c): call(void C.incI(int)) && target(c)
        && args(x)
         {
         System.out.println("Before Calling IncI");
        System.out.println(thisJoinPoint.getSignature());
           if (c.getI()+x > MAX)
                throw new RuntimeException();
```

```
 }

 before()  : call (int getI()) {
   System.out.println("Before calling getI");
   System.out.println(thisJoinPoint.getSignature());
}

after()  : call (int getI())  {
   System.out.println("After calling getI");
   System.out.println(thisJoinPoint.getSignature());

     }

before()  : execution (int getI())  {
   System.out.println("Before executing getI");
   System.out.println(thisJoinPoint.getSignature());
}

after()  : execution (int getI())  {
   System.out.println("After executing getI");
   System.out.println(thisJoinPoint.getSignature());
}

after(int x, C c): call (void incI(int)) && target(c) && args(x)
{
   System.out.println("After Calling IncI");
   System.out.println(thisJoinPoint.getSignature());
 }

 before(int x, C c): execution (void incI(int)) && target(c) &&
args(x)
{
    System.out.println("Before Executing IncI");
   System.out.println(thisJoinPoint.getSignature());
}

after(int x, C c): execution (void incI(int )) && target(c) &&
args(x) {
    System.out.println("After Executing IncI");
   System.out.println(thisJoinPoint.getSignature());
     }
  }
```

3 CONCLUSIONS

We propose a set of refactorings that can be used to restructure the AspectJ code. The identified refactorings were tested to see if the behaviour of the program remains the same. The refactorings did preserve the behaviour of the program. By restructuring the code the quality of the code is enhanced and modular applications can be developed.These refactorings have been derived based on the existing literature in this domain.

REFERENCES

[1] Deepak Dahiya, R. K. (2006). MOVING FROM AOP TO AOSD DESIGN LANGUAGE. International Journal of Computer Science and Network Security .

[2] Dr S.A.M.Rizvi, Z. K. (2008). Introduction of Aspect Oriented Techniques for refactoring legacy software. International Journal of Computer Applications.

[3] Eli Tilevich, Y. S. (2005). Binary Refactoring: Improving Code Behind the Scenes. ACM .

[4] Kulkarni, K. S. (2010). Modularization of Enterprise Application Security Through Spring AOP. International Journal of Computer Science & Communication .

[5] Leonardo Cole, P. B. Deriving Refactorings for AspectJ. OOPSLA, (p. 2004).

[6] Lerner, M. S. (2007). Beyond Refactoring: A Framework for Modular Maintenance of Crosscutting Design Idioms.

[7] Lili He, †. a. (2006). Aspect Mining Using Clustering and Association Rule Method. IJCSNS International Journal of Computer Science and Network Security .

[8] Liu, S. A. (2006). On the Notion of Functional Aspects in Aspect-Oriented Refactoring. Workshop on Aspects, Dependencies and Interactions, France.

[9] Lodewijk Bergmans, E. E. (2008). Software Engineering Properties of Languages and Aspect Technologies. Seventh International Conference on Aspect-Oriented Software Development.

[10] Michael Mortensen, S. G. (2010). Aspect-Oriented Refactoring of Legacy Applications: An Evaluation. IEEE Transactions on Software Enginering .

[11] Miguel Pessoa Monteiro, J. M. (2004). Refactoring Object-Oriented Systems with Aspect-Oriented Concepts. Ph.D. progress report.

[12] Miguel Pessoa Monteiro, J. M. (2003). Some Thoughts On Refactoring Objects to Aspects. AOSD .

[13] Santiago A. Vidal, E. S. (2009). Aspect Mining meets Rule-based Refactoring. ACM .

[14] Steve Counsell, H. H. (2010). An Empirical Investigation of Code Smell 'Deception' and Research Contextualisation through Paul's Criteria. Journal of Computing and Information Technology - CIT 18

[15] Tom Mens, T. T. (2004). A Survey of Software Refactoring. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, .