2003－SE－142 (6)
2003／5／30

Coupling Measurement in Aspect-Oriented Systems

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

Coupling is an internal software attribute that can be used to indicate the degree of interdependence among the components of a software system. Coupling is thought to be a desirable goal in software construction, leading to better values for external attributes such as maintainability, reusability, and reliability. Aspect-oriented software development (AOSD) is a new technique to support separation of concerns in software development. In aspect-oriented systems, the basic components are aspects or classes, which consist of attributes (aspect or class instance variables) and those modules such as advice, introduction, pointcuts, and methods. Thus, in aspect-oriented systems, the coupling is mainly about the degree of interdependence among aspects and/or classes. To test this hypothesis, good coupling measures for aspect-oriented systems are needed. In this paper, we propose a coupling measure suite for assessing the coupling in aspect-oriented systems. To this end, we first identify four types of interactions between aspects and classes, and then based on these interactions, we define some coupling measures between aspects and classes.

1 Introduction

Aspect-oriented software development (AOSD) is a new technique to support separation of concerns in software development [2, 8, 10, 11]. The techniques of AOSD make it possible to modularize crosscutting aspects of a system. Like objects in object-oriented software development, aspects in AOSD 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 AOSD is focused on problem analysis, software design, and implementation techniques. However, efficient evaluations of this new design technique in a rigorous and quantitative fashion is still ignored during the current stage of the technical development. For example, it has been frequently claimed that applying an AOSD method will eventually lead to quality software, but unfortunately, there is little data to support such claim. Aspect-oriented software is supposed to be easy to maintain, reuse, and evolution, yet few quantitative studies on maintenance, reuse, and evolution have been conducted, and measures to quantify the amount of maintenance, reuse, and evolution in aspect-oriented software are lacking. In order to verify claims concerning the maintainability, reusability, and reliability of software developed using aspect-oriented techniques, software measurement tools are required.

As with object-oriented systems, we would like to be able to relate aspect-oriented structural quality to critical maintainability, reusability, and re-

liability process attributes. We need appropriate measures of aspect-oriented structure to begin to relate structure to process. The development of measures of structure appropriate to aspect-oriented software has just begun. One example is the work of Zhao who developed a suite of structure measures which are specifically designed to quantify the information flows in aspect-oriented software [12].

Coupling and cohesion are two structural attributes whose importance is well-recognized in the software engineering community. In this paper we focus on coupling; cohesion measurement for aspect-oriented systems has been studied in [13]. Coupling is an internal software attribute that can be used to indicate the degree of interdependence among the components of a software system. It has been recognized that good software design should obey the principle of low coupling. A system that has strong coupling may make it more complex because it is difficult to understand, change, and correct highly interrelated components in the system. Coupling is therefore considered to be a desirable goal in software construction, leading to better values for external attributes such as maintainability, reusability, and reliability. Recently, many coupling measures and several guidelines to measure coupling of a system have been developed for procedural software and object-oriented software [5, 7, 4, 3].

An aspect-oriented system may contain many aspects and classes. These aspects and classes are not independent; they can interact with each other in various kinds of ways. Thus, in aspect-oriented systems, the coupling is mainly about the degree of interdependence among aspects and/or classes. To test this hypothesis, good coupling measures for aspect-oriented systems are needed. Moreover, in order to measure the couping of an aspect-oriented system, we should consider various types of interactions between aspects and classes in the system.

However, although coupling has been widely studied for procedural and object-oriented software [4, 3, 5, 7], it has not been studied for aspect-oriented software yet. Moreover, existing approaches to measuring the coupling of procedural and object-oriented software can not be directly applied to aspect-oriented systems since it contains new types of interactions that are different from those in object-oriented software, new program representations that are appropriate for representing the interactions between these new modules are needed.

In this paper, we propose a coupling measure suite for assessing the coupling in aspect-oriented systems. To this end, we first identify four types of interactions between aspects and classes, and then based on these interactions, we define some coupling measures between aspects and classes.

Because aspect-oriented paradigm significantly different from procedural and object-oriented paradigms, we really need to develop a notion of coupling for aspect-oriented systems, which is an indicator of the degree to which the components in the system interact each other. We hope that by examining the ideas of the coupling in aspect-oriented systems from several different viewpoints and through independently developed measures, we can have a better understanding of what the coupling is meant in aspect-oriented systems and the role that coupling plays in the development of quality aspect-oriented software. As the first step to study the coupling in aspect-oriented systems, in this paper we would like to provide a sound and formal basis for coupling measurement in aspect-oriented systems before applying it to real aspect-oriented system design.

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 identifies four types of interactions between aspects and classes in an aspect-oriented system. Section 4 proposes some coupling measures for aspect-oriented systems. Section 5 disscusses some related work, and concluding remarks are given in Section 6.

```
 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.

2 Aspect-Oriented Programming with AspectJ

We present our coupling measures in the context of AspectJ, the most widely used aspect-oriented programming language [9]. Our basic techniques, however, deal with the basic concepts of aspect-oriented programming and therefore apply to the general class of AOP languages.

AspectJ [9] is a seamless aspect-oriented extension to Java; AspectJ adds some new concepts and associated constructs to Java. These concepts and associated constructs are called join points, pointcut, advice, introduction, and aspect. We briefly introduce each of these constructs as follows.

The *aspect* is the modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that crosscuts other classes in a program. Like a class, an aspect can be instantiated, can contain state and methods, and also may be specialized with subaspects. An aspect is combined with the classes it crosscuts according to specifications given within the aspect. Moreover, an aspect can use an *introduction* construct to introduce methods, attributes, and interface implementation declarations into classes. 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 elements. For example, the aspect `PointShadowProtocol` in Figure 1 privately introduces a field `shadow` to the class `Point` at `s31`.

A central concept in the composition of an aspect 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, an 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 example, the aspect `PointShadowProtocol` in Figure 1 declares three pointcuts named `setting`, `settingX`, and `settingY` at `p36`, `p37`, and `p38`.

An aspect can specify *advice*, which is used to define code that executes when a pointcut is reached. Advice is a method-like mechanism which consists of instructions that execute *before*, *after*, or *around* a pointcut. `around` advice executes *in place* of the indicated pointcut, allowing a method to be replaced. For example, the aspect `PointShadowProtocol` in Figure 1 declares three pieces of after advice at `ae39`, `ae43`, and `ae48`; each is attached to the corresponding pointcut `setting`, `settingX`, or `settingY`.

An AspectJ program can be divided into two parts: *base code* which includes classes, interfaces, and other standard Java constructs and *aspect code* which implements the crosscutting concerns in the program. For example, Figure 1 shows an AspectJ program that associates shadow points with every `Point` object. The program can be divided into the base code containing the classes `Point` and `Shadow`, and the aspect code which has the aspect `PointShadowProtocol` that stores a shadow object in every `Point`. Moreover, the AspectJ implementation ensures that the aspect and base code run together in a properly coordinated fashion. The key component is the *aspect weaver*, when ensures that applicable advice runs at the appropriate join points. For more information about AspectJ, refer to [1].

To focus on the key ideas of our work, we do not discuss the coupling between classes in this paper because they can be measured using existing techniques [3, 7].

Example. Figure 1 shows an AspectJ program taken from [1] that associates shadow points with every `Point` object. 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`. Through the rest of the paper, We use this example program to demonstrate our basic idea of coupling measurement.

In the rest of the paper, we assume that an aspect is composed of attributes (aspect instance variables), and modules[2] such as advice, introduction, pointcuts and methods.

3 Aspect-Class Interactions

In an aspect-oriented system, an aspect can interact with a class in several ways, i.e., by *object creation*, *method call*, *introduction declaration*, and *join point*. These types of interactions between aspects and classes may affect the coupling between classes and aspects.

3.1 Object-Creation Interactions

The object-creation interaction is related to the case that a module m in an aspect α may create an object of a class C through a declaration or by using an operator such as `new`. At this time, there is an implicit call from m to C's constructor. An object-creation interaction can be denoted as *O-interaction*. For example, statement `s40` in Figure 1 represents an object creation of class `Shadow` in aspect `PointShadowProtocol`.

[1] Unlike a method that has a unique method 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.

[2] For unification, we use the word a "module" to stand for a piece of advice, a piece of introduction, a pointcut, or a method declared in an aspect.

3.2 Method-Call Interactions

Method-call interactions may occur when an aspect α may have a call from its module m_1 to a method m_2 in the public interface of class C. A method-call interaction can be denoted as *M-interaction*. For example, statement `s45` in Figure 1 represents a call to method `getX()` of class `Point` in aspect `PointShadowProtocol`.

3.3 Introduction Interactions

Introduction interactions may exist when an aspect α declares a piece of introduction i that publicly introduces one method (or constructor) into a class C. If a piece of introduction in α publicly introduces a field *fd* into a class C, we regard *fd* as an instance variable of both α and C. Therefore, *fd* is accessible to all modules in α and C. If a piece of introduction in α privately introduces a field *fd* into a class C, *fd* is only accessible to all modules in α. An introduction interaction can be denoted as an *I-interaction*. For example, statement `s31` in Figure 1 represents a piece of introduction that privately introduce a field `shadow` into class `Point`; this means only code defined in `PointShadowProtocol` can access `shadow`.

3.4 Join-Point Interactions

In an aspect-oriented system, an aspect may be woven into one or more classes at some join points, declared within *pointcuts* which are used in the definition of *advice*. This leads to that a piece of advice in an aspect may advise one or more methods in a class. By carefully examining the pointcuts and their associated advice, one can determine those methods that a piece of advice may advise. This information can be used to connect the base program (classes) and aspects. A join-point interaction can be denoted as *J-interaction*. For example, the after advice declared in aspect `PointShadowProtocol` (lines ae43–s47) of Figure 1 may weave into method `setX()` of class `Point`.

4 Coupling Measures Based on Aspect-Class Interactions

We next present a coupling measure suite for aspect-oriented systems based on the four types of interactions between aspects and classes discussed previously.

4.1 O-Interaction Based Coupling Measure

We define this measure related to coupling between aspects and classes, denoted by OICM as follows. The OICM for an aspect is a count of the number of classes to which it is interacted due to object creations.

4.2 M-Interaction Based Coupling Measure

We define this measure related to coupling between aspects and classes, denoted by MICM as follows. The MICM for an aspect is a count of the number of classes to which it is interacted due to method calls.

4.3 I-Interaction Based Coupling Measure

We define this measure related to coupling between aspects and classes, denoted by IICM as follows. The IICM for an aspect is a count of the number of classes to which it is interacted due to introduction.

4.4 J-Interaction Based Coupling Measure

We define this measure related to coupling between aspects and classes, denoted by JICM as follows. The JICM for an aspect is a count of the number of classes to which it is interacted due to join points.

5 Related work

We discuss some related work in the area of measurement of aspect-oriented systems. To the best of our knowledge, our work is the first attempt to study how to assess the coupling between aspects and classes in aspect-oriented systems.

Zhao [12] proposes a metrics suite for aspect-oriented software, which are specifically designed to quantify the information flows in an aspect-oriented program. To this end, He presents a dependence model for aspect-oriented software which is composed of several dependence graphs to explicitly represent dependence relationships in a module, a class, or the whole program. Based on the model, Zhao defines various kinds of metrics that can be used to measure the complexity of an aspect-oriented program from various different viewpoints. However, although Zhao's approach can assess the complexity of aspect-oriented software from various different viewpoints, it does not address the issue related to coupling measurement in aspect-oriented software.

6 Concluding Remarks

In this paper, we proposed a coupling measure suite for assessing the coupling in aspect-oriented systems. To this end, we first identified four types of interactions between aspects and aspects, and then based on these interactions, we defines some coupling measures between aspects and classes.

The coupling measures proposed in this paper focused only on the interactions between aspects and classes, and did not take the interactions between aspects or classes into account. These issues need also to consider in order to build a whole coupling measurement suite. Also, in this paper we did not consider to measure the coupling due to derived aspects (i.e., aspect inheritance). In our future work, we will study the influence of aspect inheritance and other aspect-oriented features on coupling, and apply our coupling measure suite to real aspect-oriented system design.

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] L.C. Briand, J. Daly and J. Wuest. A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering, Vol.25, No.1, pp.91-121.

[4] L.C. Briand, P. Devanbu, and W. Melo, "An Investigation into Coupling Measures for C++," Proc. 19th International Conference on Software Engineering, pp.412-421, May 1997.

[5] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, pp.476-493, Vol.20, No.6, 1994.

[6] B. Henderson-Sellers. Software Metrics. Prentice Hall, Hemel Hempstaed, U.K., 1996.

[7] M. Hitz and B. Montazeri. Measuring Coupling and Cohesion in Object-Oriented Systems. *Proceedings of International Symposium on Applied Corporate Computing*, pp.25-27, Monterrey, Mexico, October 1995.

[8] 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.

[9] 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*, pp220-242, LNCS, Vol.1241, Springer-Verlag, June 2000.

[10] 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.

[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. Zhao, "Toward A Metrics Suite for Aspect-Oriented Software," Technical Report SE-136-5, Information Processing Society of Japan (IPSJ), March 2002.

[13] J. Zhao and B. Xu, "Cohesion Measurement for Aspect-Oriented Systems," March 2003. (submitted for publication)