社団法人 情報処理学会　研究報告
IPSJ SIG Technical Report

2004－SE－144　(27)
2004／3／19

Cohesion Measures for Aspect-Oriented Programs

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

Cohesion is an internal software attribute that can be used to indicate how tightly the modules of a software component are bound together in design and implementation. Cohesion 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. AOSD introduces a new kind of component called aspect which is like a class, also consisting of attributes (aspect instance variables) and those modules such as advice, introduction, pointcuts, and methods. The cohesion for such an aspect is therefore about how tightly the attributes and modules of aspects cohere. To test this hypothesis, cohesion measures for aspects are needed. In this paper, we propose an approach to assessing the aspect cohesion based on dependence analysis. To this end, we present various types of dependencies between attributes and/or modules in an aspect, and the aspect dependence graph (ADG) to explicitly represent these dependencies. Based on the ADG, we formally define some aspect cohesion measures. We also discuss the properties of these dependencies, and according to these properties, we prove that these measures satisfy the properties that a good measure should have.

1 Introduction

Aspect-oriented software development (AOSD) is a new technique to support separation of concerns in software development [2, 11, 12, 13]. The techniques of AOSD make it possible to modularize crosscutting aspects of a system. 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 are 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 evolve, yet few quantitative studies have been conducted, and measures to quantify the amount of maintenance, reuse, and evolution in aspect-oriented systems are lacking. In order to verify claims concerning the maintainability, reusability, and reliability of systems developed using aspect-oriented techniques, software measures are required.

As with procedural and object-oriented systems, we would like to be able to relate aspect-oriented structural quality to critical maintainability, reusability, and reliability 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 dependence-based structural measures which are specifically designed to quantify the information flows in aspect-oriented software [14], and also a coupling measures suite for aspect-oriented software [16].

Cohesion is a structural attribute whose importance is well-recognized in the software engineering community. It is an internal software attribute that can be used to indicate how tightly the modules of a software component are bound together in design or implementation. In procedural or object-oriented paradigm, a highly cohesive component is one with one basic function and it should be difficult to decompose a cohesive component. Cohesion is therefore considered to be a desirable goal in software construction, leading to better values for external attributes such as maintainability, reusability, and reliability. A system should have high cohesion. Recently, many cohesion measures and several guidelines to measure cohesion of a component have been developed for procedural software [3, 10] and for

object-oriented software [6, 8, 9, 4, 5].

Aspect-oriented language introduces a new kind of component called *aspect* to model the crosscutting concerns in a software system. An aspect with its encapsulation of state (attributes) and associated modules (operations) such as advice, introduction, pointcuts, and methods is a different abstraction in comparison to a procedure within procedural systems and a class within object-oriented systems. The cohesion of an aspect is therefore mainly about how tightly the aspect's attributes and modules cohere. However, although cohesion has been studied widely for procedural and object-oriented software, it has not been studied for aspect-oriented software yet. Since an aspect contains new modules such as advice, introduction, and pointcuts that are different from methods in a class, existing class cohesion measures can not be directly applied to aspects. Therefore, new measures that are appropriate for measuring aspect cohesion are needed.

In this paper, we propose an approach to assessing the aspect cohesion based on dependence analysis. To this end, we present various types of dependencies between attributes and/or modules such as advice, introduction, pointcuts, and methods of an aspect, and a dependence-based representation called *aspect dependence graph* (ADG) to represent these dependencies. Based on the ADG, we formally define some aspect cohesion measures. We also discuss the properties of these dependencies, and according to these properties, we prove that these measures satisfy the properties that a good measure should have.

We hope that by studying the ideas of aspect cohesion from several different viewpoints and through well developed cohesion measures, we can obtain a better understanding of what the cohesion is meant in aspect-oriented paradigm and the role that cohesion plays in the development of quality aspect-oriented software. As the first step to study the aspect cohesion, the goal of this paper is to provide a sound and formal basis for aspect cohesion measures before applying them to real aspect-oriented software 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 defines three types of dependencies in an aspect and discusses some basic properties of these dependencies. Section 4 proposes an approach to measuring aspect cohesion from three facets: inter-attribute, module-attribute and inter-module. Section 5 discusses some related work. Concluding remarks are given in Section 6.

2 Aspect-Oriented Programming and AspectJ

We present our basic ideas of aspect cohesion measurement approach for aspect-oriented programs in the context of AspectJ, the most widely used aspect-oriented programming language [1]. Our basic ideas, 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 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 1 shows an AspectJ program that associates shadow points with every `Point` object. The program contains one aspect `PS_Protocol` and two classes

```
public class Point {
  protected int x, y;
  public Point(int _x, int _y) {
    x = _x;
    y = _y;
  }
  public int getX() {
    return x;
  }
  public int getY() {
    return y;
  }
  public void setX(int _x) {
    x = _x;
  }
  public void setY(int _y) {
    y = _y;
  }
  public void printPosition() {
    System.out.println("Point
                  at("+x+","+y+")");
  }
  public static void main(String[] args) {
    Point p = new Point(1,1);
    p.setX(2);
    p.setY(2);
  }
}
class Shadow {
  public static final int offset = 10;
  public int x, y;

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

```
aspect PS_Protocol {
  private int shadowCount = 0;
  public static int getCount() {
    return PS_Protocol.aspectOf().shadowCount;
  }
  private Shadow Point.shadow;
  public static void associate(Point p, Shadow s){
    p.shadow = s;
  }
  public static Shadow getShadow(Point p) {
    return p.shadow;
  }

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

  after(int x, int y, Point p) returning :
    setting(x, y, p) {
    Shadow s = new Shadow(x,y);
    associate(p,s);
    shadowCount++;
  }
  after(Point p): settingX(p) {
    Shadow s = new getShadow(p);
    s.x = p.getX() + Shadow.offset;
    p.printPosition();
    s.printPosition();
  }
  after(Point p): settingY(p) {
    Shadow s = new getShadow(p);
    s.y = p.getY() + Shadow.offset;
    p.printPosition();
    s.printPosition();
  }
}
```

Figure 1. A sample AspectJ program.

`Point` and `Shadow`. The aspect has three methods `getCount`, `associate` and `getShadow`, and three pieces of advice related to pointcuts `setting`, `settingX` and `settingY` respectively[1]. The aspect also has two attributes, i.e., `shadowCount` which is an attribute of the aspect itself and `shadow` which is an attribute that is privately introduced to class `Point`.

In the rest of the paper, we use this example to demonstrate our basic ideas of aspect cohesion measurement. We also assume that an aspect is composed of attributes (aspect instance variables), and modules[2] such as advice, introduction, pointcuts and methods.

3 Aspect Dependencies

In this section we define various types of dependencies between modules and/or attributes in an aspect and discuss some properties of these dependencies.

[1]Since 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 "module" to stand for a piece of advice, a piece of introduction, a pointcut, or a method declared in an aspect.

3.1 Dependence Definitions

We define three types of dependencies between attributes and/or modules in an aspect, that is, *inter-attribute*, *inter-module*, and *module-attribute dependence*.

Definition 1 *Let a_1, a_2 be attributes in an aspect. a_2 is* inter-attribute-dependent *on a_1, denoted by $a_2 \hookrightarrow a_1$, if one of the following conditions holds:*

- *The definition of a_2 uses (refers) a_1 directly or indirectly, or*
- *Whether a_2 can be defined is determined by the state of a_1.*

Generally, if a_2 is used in the condition part of a control statement (such as `if` and `while`), and the definition of a_1 is in the inner statement of the control statement, then the definition of a_1 depends on the state of a_2. For example, according to Definition 1, we know that there is no inter-attribute dependence in aspect `PS_Protocol` of Figure 1.

There are two types of dependencies between aspect modules: *inter-module call dependence* and *inter-module potential dependence*.

Definition 2 *Let m_1, m_2 be two modules and a be an attribute in an aspect. m_2 is* inter-module-dependent *on m_1,*

denoted by $m_2 \rightarrow m_1$, if one of the following conditions holds:

- *m_1 is called in m_2. (inter-module call dependence)*
- *a is used in m_2 before it is defined, and a is defined in m_1. (inter-module potential dependence)*

Given an aspect, we can not assume which piece of introduction or which method in the aspect might be invoked before another[1]. So we assume that all the introduction and methods in the aspect can be invoked at any time and in any order. Therefore, if m_2 might use an attribute a, and a is defined in m_1, and if m_1 is invoked first and then m_2 is invoked, then m_2 might use a defined in m_1, i.e. m_2 is inter-module potentially-dependent on m_1.

To obtain the inter-module dependencies, for each module m, we introduce two sets, D_{IN} and D_{OUT}, where $D_{IN}(m)$ is the set of attributes referred before modifying their values in m, and $D_{OUT}(m)$ is the set of attributes modified in m. Thus, for an attribute a and modules m and m', if $a \in D_{IN}(m') \cap D_{OUT}(m)$, then $m' \rightarrow m$.

In addition to attributes, since there are four types of different modules in an aspect, i.e., advice, introduction, pointcuts and methods, there may have the following possible types of inter-module dependencies, i.e., dependencies between advice and advice, advice and introduction, advice and method, advice and pointcut[2], introduction and introduction, introduction and method, or method and method.

Example. In order to compute inter-module dependencies in aspect `PS_Protocol`, we first compute the D_{IN} and D_{OUT} sets for each module in `PS_Protocol`. They are: D_{IN}(`getCount`) = {`shadowCount`}, D_{OUT}(`getCount`) = $\emptyset$, D_{IN}(`getShadow`) = {`shadow`}, D_{OUT}(`getShadow`) = $\emptyset$, D_{IN}(`associate`) = $\emptyset$, D_{OUT}(`associate`) = {`shadow`}, D_{IN}(`setting`) = {`shadowCount`}, D_{OUT}(`setting`) = {`shadowCount`}, D_{IN}(`settingX`) = D_{OUT}(`settingX`) = $\emptyset$, D_{IN}(`settingY`) = D_{OUT}(`settingY`) = $\emptyset$.

Since there also exist inter-module dependencies between each pointcut and its corresponding advice, we finally get the following inter-module dependencies in `PS_Protocol`.

(method `getCount` → advice`setting`), (advice `setting` → method `associate`), (advice `settingX` → method `getShadow`), (advice `settingY` → advice `getShadow`), (pointcut `setting` → advice `setting`), (pointcut `settingX` → advice `settingX`), and (pointcut `settingY` → advice `settingX`).

Definition 3 *Let m be a module and a be an attribute in an aspect. m is* module-attribute-dependent *on a, denoted by $m \mapsto a$, if a is referred in m.*

Since there are four types of different modules in an aspect, module-attribute dependencies may have four different types: *advice-attribute*, *introduction-attribute*, *pointcut-attribute*, or *method-attribute* dependencies.

[1] In AspectJ, advice is automatically woven into some methods in a class by the compiler, and therefore no call exists for the advice.

[2] A pointcut is only related to its corresponding advice. Therefore, there is no dependence between pointcut and method, pointcut and introduction, or pointcut and pointcut.

Example. According to Definition 3, the module-attribute dependencies in aspect `PS_Protocol` are: (method `getCount` ↦ attribute `shadowCount`), (method `getShadow` ↦ attribute `shadowCount`), (advice `settingX` ↦ attribute `shadowCount`).

Note that all these types of dependencies defined above can be derived by performing control flow and data flow analysis using existing flow analysis algorithms [15]. Due to the space limitation, we do not discuss this issue here.

3.2 Dependence Properties

This section discusses some properties of dependencies defined in Section 3.1, and refines the definition of inter-module dependence to fit for measuring aspect cohesion.

According to definition 1, if $a_1 \hookrightarrow a_2$ and $a_2 \hookrightarrow a_3$, then $a_1 \hookrightarrow a_3$. Therefore, we have

Property 1 *The inter-attribute dependencies are transitive.*

Based on Property 1, we can define the inter-attribute transitive dependence as follows.

Definition 4 *Let A be an aspect and a_i $(i > 0)$ be attributes in A. If there exist attributes $a_1, a_2, \ldots, a_{n-1}, a_n$ $(n > 1)$, where $a_1 \hookrightarrow a_2, \ldots, a_{i-1} \hookrightarrow a_i, \ldots, a_{n-1} \hookrightarrow a_n$, then a_1 is* inter-attribute-transitive-dependent *on a_n, denoted by $a_1 \overset{*}{\hookrightarrow} a_n$.*

According to definition 2, for modules m_1, m_2, and m_3, if $m_1 \rightarrow m_2$ and $m_2 \rightarrow m_3$, then $m_1 \rightarrow m_3$ may not hold. Consider an example of inter-module call dependence, if m_1 is called in m_2 and m_2 is called in m_3, then m_1 is not necessarily called in m_3. For inter-module potential dependence, if $m_1 \rightarrow m_2$ and $m_2 \rightarrow m_3$ are introduced by unrelated, different attributes, then m_1 might have no relation with m_3. Therefore, we have

Property 2 *The inter-module dependencies are not transitive.*

The intransitivity among inter-module dependencies leads to great difficulties when performing analysis. Thus, we should redefine the inter-module dependencies.

Definition 5 *Let m_1, m_2 be modules and a be an attribute in an aspect. If a is used in m_1 and defined in m_2, then a used in m_1 is dependent on a defined in m_2, denoted by $m_1 \xrightarrow{a,a} m_2$, where <a, a> is named as a tag.*

For unification, add a tag <*, *> for each inter-module call dependence arc, i.e., if m_1 is inter-module-call-dependent on m_2, then we have $m_1 \xrightarrow{*,*} m_2$.

Definition 5 is the basic definition. Since the dependencies between attributes are transitive, we can obtain a more general definition according to Property 3.

To obtain such dependencies, we introduce two sets for each module m of an aspect, i.e., D_A and D_{AO}, each element of which has the form (a, a'), where a and a' are attributes of the aspect.

- $D_A(m)$ is the set of dependencies which records the dependencies from the attributes referred in m to the attributes defined out m. $D_A(m)$ is a subset of inter-attribute dependencies.

- $D_{AO}(m)$ is the set of dependencies which records the dependencies from the attributes referred in m to the attribute defined out m when exiting m.

In general, the intermediate results are invisible outside, and an attribute might be modified many times in a piece of advice, a piece of introduction, or a method. We introduce D_{AO} to improve the precision. Obviously, we have $D_{AO}(m) \subseteq D_A(m)$.

Definition 6 *Let m_1, m_2 be modules and a_1, a_2 be attributes in an aspect. If $(a_1, a_2) \in D_A(m_1)$ and $a_2 \in D_{OUT}(m_2)$, then m_1 is dependent on m_2, denoted by $m_1 \stackrel{a_1,a_2}{\longrightarrow} m_2$.*

According to Definition 6, we have the following properties:

Property 3 *Let m_1, m_2, m_3 be modules and a_1, a_2, a_3 be attributes in an aspect. If $m_1 \stackrel{a_1,a_2}{\longrightarrow} m_2$, and $\forall(a_2, a_3)$, $(a_2, a_3) \in D_{AO}(m_2)$ and $a_3 \in D_{OUT}(m_3)$, then $m_1 \stackrel{a_1,a_3}{\longrightarrow} m_3$.*

Because $D_{AO}(m_1) \subseteq D_A(m_1)$, according to Definition 6, if $(a_2, a_3) \in D_{AO}(m_1)$, and $a_3 \in D_{OUT}(m_2)$, then $m_1 \stackrel{a_1,a_2}{\longrightarrow} m_3$. Thus, we have Corollary 1.

Corollary 1 *Let m_1, m_2, m_3 be modules and a_1, a_2, a_3 be attributes in an aspect. If $m_1 \stackrel{a_1,a_2}{\longrightarrow} m_2$ and $m_2 \stackrel{a_2,a_3}{\longrightarrow} m_3$, then $m_1 \stackrel{a_1,a_3}{\longrightarrow} m_3$.*

Property 4 *Let m_1, m_2, m_3 be modules and a_1, a_2 be attributes in an aspect. If $m_1 \stackrel{*,*}{\longrightarrow} m_2$ and $m_2 \stackrel{a_1,a_2}{\longrightarrow} m_3$, then $m_1 \stackrel{a_1,a_2}{\longrightarrow} m_3$.*

From Properties 2-4, we can define the inter-module transitive dependence as follows.

Definition 7 *Let A be an aspect, m_i $(i > 0)$ be modules, and a_i $(i > 0)$ be attributes in A. If there exist modules $m_1, \ldots, m_n$ and attributes $a_1, \ldots, a_n$ $(n > 1, a_i$ need not be unique, and a_i may be "$*$", which models calls between modules), where $m_1 \stackrel{a_1,a_2}{\longrightarrow} m_2, \ldots, m_{i-1} \stackrel{a_{i-1},a_i}{\longrightarrow} m_i, \ldots, m_{n-1} \stackrel{a_{n-1},a_n}{\longrightarrow} m_n$, then m_1 is* inter-module-transitive-dependent *on m_n, denoted by $m_1 \stackrel{*}{\longrightarrow} m_n$.*

To present our cohesion measure in a unified model, we introduce the aspect dependence graph to explicitly represent all types of dependencies in an aspect.

Definition 8 *The aspect dependence graph (ADG) of an aspect A is a directed graph[3], $G_{ADG} = (V, A, T)$ where $V = V_a \cup V_m$, $A = A_{aa} \cup A_{mm} \cup A_{ma}$, and $T \in A \times (V', V')$ are the sets of vertex, arc, and tag respectively, such that:*

- *V_a is the set of attribute vertices: each represents a unique attribute (the name of a vertex is the name of the attribute it represents) in A.*

- *V_m is the set of module vertices: each represents a unique module (the name of a vertex is the name of the module it represents) in A.*

- *V' is the union of V_a and $\{*\}$, i.e., $V' = V_a \cup \{*\}$.*

- *A_{aa} is the set of inter-attribute dependence arcs that represents dependencies between attributes, such that for v_a, $v_{a'} \in V_a$ if $a \hookrightarrow a'$, then $(v_a, v_{a'}) \in A_{aa}$.*

- *A_{mm} is the set of inter-module dependence arcs that represent dependencies between modules, such that for v_m, $v_{m'} \in V_m$ if $m \to m'$, then $(v_m, v_{m'}) \in A_{mm}$.*

- *A_{ma} is the set of module-attribute dependence arcs that represents dependencies between modules and attributes, such that for $v_m \in V_m$, $v_a \in V_a$ if $m \mapsto a$, then $(v_m, v_a) \in A_{ma}$.*

Generally, G_{ADG} consists of three sub-graphs, i.e., *inter-attribute dependence graph* $G_{AAG} = (V_a, A_{aa})$, *inter-module dependence graph* $G_{MMG} = (V_m, A_{mm}, T)$, and *module-attribute dependence graph* $G_{MAG} = (V, A_{ma})$, which can be used to define the *inter-attribute*, *inter-module*, and *module-attribute* cohesion in an aspect respectively. Figure 2 shows the G_{MAG} and G_{MMG} of aspect `PS_Protocol`. Since there exists no inter-attribute dependence in the aspect, the G_{AAG} is not available. Note that we omit the Tags in both graphs for convenience.

4 Measuring Aspect Cohesion

Briand *et al.* [4] have stated that a good cohesion measure should have properties such as *non-negative and standardization*, *minimum and maximum*, *monotony*, and *cohesion does not increase when combining two components*. We believe that these properties provide also a useful guideline even in aiding the development of an aspect cohesion measure. In this section, we propose our aspect cohesion measures, and show that our aspect cohesion measures satisfy the properties given by Briand *et al.* [4].

[3] A directed graph $G = (V, A)$, where V is a set of vertices and $A \in V \times V$ is a set of arcs. Each arc $(v, v') \in A$ is directed from v to v'; we say that v is the source and v' the target of the arc.

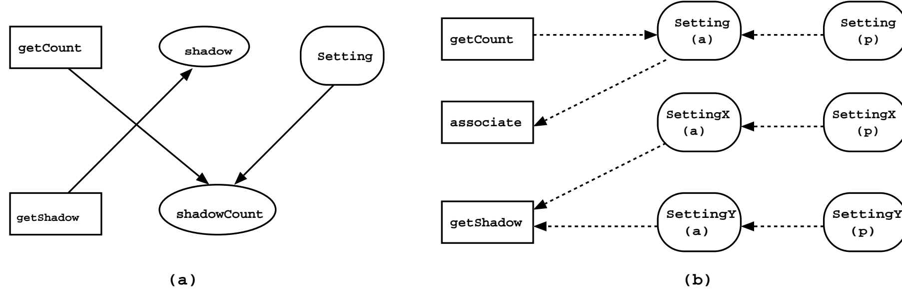

Figure 2. The G_{MAG} (a) and G_{MMG} (b) of the aspect PS_Protocol in Figure 1.

An aspect consists of attributes and modules such as advice, introduction, pointcuts, and methods. There are three types of dependencies between attributes and/or modules. Thus, the cohesion of an aspect should be measured from the three facets. In the following discussion, we assume that an aspect A consists of k attributes and n modules, where $k, n \geq 0$.

4.1 Measuring Inter-Attribute Cohesion

Inter-attribute cohesion is about the tightness between attributes in an aspect. To measure the inter-attribute cohesion for an aspect A, for each attribute a of A, we introduce a set D_a to record the attributes on which a depends, i.e., $D_a(a) = \{a \mid a_1 \overset{*}{\hookrightarrow} a, a_1 \neq a\}$. Thus, we define the inter-attribute cohesion of A as:

$$\gamma_a(A) = \begin{cases} 0 & \text{k = 0} \\ 1 & \text{k = 1} \\ \frac{1}{k}\sum_{i=1}^{k} \frac{|D_a(a_i)|}{k-1} & k > 1 \end{cases}$$

where k is the number of attributes in A, and $\frac{|D_a(a_i)|}{k-1}$ represents the degree on which a_i depends on other attributes in A.

If $k = 0$, there is no attribute in A. Inter-attribute cohesion is useless, thus we set $\gamma_a(A) = 0$. If $k = 1$, there is only one attribute in A. Although it can not depend on other attribute, it itself is tight, thus we set $\gamma_a(A) = 1$. If each attribute relates to all others, then $\gamma_a(A) = 1$. If all attributes can exist independently, then $\gamma_a(A) = 0$. Thus, $\gamma_a(A) \in [0, 1]$.

Theorem 1 *Let A be an aspect and $G_{AAG} = (V_a, A_{aa})$ be the inter-attribute dependence graph of A. $\gamma_a(A)$ does not decrease when adding an arc $(a_1, a_2) \in A_{aa}$, where $a_1, a_2 \in V_a$, on G_{AAG}.*

Theorem 2 *Let A_1 and A_2 be two aspects and A_{12} be an aspect derived from the combination of A_1 and A_2. Let $\gamma_a(A_1)$ and $\gamma_a(A_2)$ be the inter-attribute cohesions of A_1 and A_2 respectively and $\gamma_a(A_{12})$ be the inter-attribute cohesion of A_{12}. $\gamma_a(A_{12}) \leq max\{\gamma_a(A_1), \gamma_a(A_2)\}$.*

Example. The D_a sets for each module in `PS_Protocol` are: $D_a(\texttt{shadowCount}) = D_a(\texttt{shadow}) = \emptyset$. Therefore, $\gamma_a(\texttt{PS_Protocol}) = \frac{1}{2}\sum_{i=1}^{2} \frac{|D_a(a_i)|}{2-1} = 0$.

4.2 Measuring Module-Attribute Cohesion

Module-attribute cohesion is about the tightness between modules and attributes in an aspect. To measure this kind of cohesion, for each module m in an aspect A, we introduce two sets: D_{ma} and D^o_{ma}, where

- $D_{ma}(m)$ is the set of all A's attributes that are referred in m.
- $D^o_{ma}(m)$ is a set of all A's attributes that are referred in m and related to attributes referred in other modules, i.e.,

  $D^o_{ma}(m) = \{a \mid \exists a_1, m_1 \text{ such that } ((m_1 \overset{a_1,a}{\longrightarrow} m) \vee (m \overset{a,a_1}{\longrightarrow} m_1)) \wedge (a, a_1 \neq '*')\}$.

Obviously, $D^o_{ma}(m) \subseteq D_{ma}(m)$. We can define the module-attribute cohesion for A as follows:

$$\gamma_{ma}(A) = \begin{cases} 0 & \text{n = 0} \\ 1 & \text{n = 1 and } |D_{ma}(m_i)| \neq 0 \\ \frac{1}{n}\sum_{i=1}^{n} \frac{|D^o_{ma}(m_i)|}{|D_{ma}(m_i)|} & \text{others} \end{cases}$$

where n is the number of modules in A, and $\frac{|D^o_{ma}(m_i)|}{|D_{ma}(m_i)|}$, denoted by $\rho(m_i)$, is the ratio between the number of attributes which are referred in m_i and relevant to others, to the number of all attributes referred in m_i.

For a module m, if $D_{ma}(m) = \emptyset$, i.e., no attribute is referred in m, we set $\rho(m) = 0$. If the attributes referred in m are not related to other modules, these attributes can work as local variables. It decreases the cohesion to take a local variable for a module as an attribute for all modules. If there is no attribute or module in the aspect, no module will depend on others. There is no D_{ma} or all the D_{ma} are empty, i.e., $|D_{ma}(m)| = 0$. Thus, $\gamma_{ma} = 0$. If each attribute referred in m is related to other modules, then $\rho(m) = 1$.

Theorem 3 *Let A be an aspect and $G_{MMG} = (V_m, A_{mm})$ be the inter-module dependence graph of A. Let m_1 be a*

module of A and $\rho(m_1) = \frac{|D^o_{ma}(m_1)|}{|D_{ma}(m_1)|}$. $\rho(m_1)$ does not decrease when adding an arc (m_1, m_2), where $m_1, m_2 \in V_m$, on G_{MMG}.

Example. The D_{ma} and D^o_{ma} sets for each module in `PS_Protocol` are: $D_{ma}(\texttt{getShadow}) = \{\texttt{shadow}\}$, $D_{ma}(\texttt{getCount}) = D_{ma}(\texttt{setting}) = \{\texttt{shadowCount}\}$, $D_{ma}(\texttt{associate}) = D_{ma}(\texttt{settingX}) = D_{ma}(\texttt{settingY}) = \emptyset$, $D^o_{ma}(\texttt{getCount}) = D^o_{ma}(\texttt{getShadow}) = \emptyset$, $D^o_{ma}(\texttt{associate}) = D^o_{ma}(\texttt{setting}) = \emptyset$, and $D^o_{ma}(\texttt{settingX}) = D^o_{ma}(\texttt{settingY}) = \emptyset$. Therefore, $\gamma_{ma}(\texttt{PS_Protocol}) = \frac{1}{6}\sum_{i=1}^{6} \frac{|D^o_{ma}(m_i)|}{|D_{ma}(m_i)|} = 0$.

4.3 Measuring Inter-Module Cohesion

In the G_{MMG}, although the modules can be connected by attributes, this is not necessary sure that these modules are related. If there does exist some relations between modules, we should determine their tightness. This is the process to measure the inter-module cohesion. To do this, we introduce another set D_m for each module m in an aspect A, where $D_m(m) = \{m_2 \mid m_1 \xrightarrow{*} m_2\}$.

The inter-module cohesion $\gamma_m(A)$ for A is defined as follows:

$$\gamma_m(A) = \begin{cases} 0 & \text{n} = 0 \\ 1 & \text{n} = 1 \\ \frac{1}{n}\sum_{i=1}^{n} \frac{|D_m(m_i)|}{n-1} & n > 1 \end{cases}$$

where n is the number of modules in A and $\frac{|D_m(m_i)|}{n-1}$ represents the tightness between m_i and other modules in A. If each module depends on all other modules, then $\gamma_m(A) = 1$. If all modules are independent, i.e., each module has no relation with any other modules, then $\gamma_m(A) = 0$.

Theorem 4 *Let A be an aspect, $G_{MMG} = (V_m, A_{mm})$ be the inter-module dependence graph of A. The inter-module cohesion $\gamma_m(A)$ does not decrease when adding an arc $(m_1, m_2) \in A_{mm}$, where $m_1, m_2 \in V_m$, on G_{MMG}.*

Theorem 5 *Let A_1 and A_2 be aspects and A_{12} be an aspect derived from the combination of A_1 and A_2. Let $\gamma_m(A_1)$ and $\gamma_m(A_2)$ be the inter-module cohesions of A_1 and A_2 and $\gamma_m(A_{12})$ be the inter-module cohesion of A_{12}. $\gamma_m(A_{12}) \leq max\{\gamma_m(A_1), \gamma_m(A_2)\}$.*

We can prove Theorems 4 and 5 with a similar way as we did for Theorems 1 and 2. Due to the limitation of the space, we do not repeat them here.

Example. The D_m sets for each module in `PS_Protocol` are: $D_m(\texttt{getCount}) = \{\texttt{setting}\}$, $D_m(\texttt{getShadow}) = \{\texttt{shadow}\}$, $D_m(\texttt{setting}) = \{\texttt{associate}\}$, $D_m(\texttt{settingX}) = \{\texttt{getShadow}\}$, and $D_m(\texttt{settingY}) = \{\texttt{getShadow}\}$. Therefore, $\gamma_m(\texttt{PS_Protocol}) = \frac{1}{6}\sum_{i=1}^{6} \frac{|D_m(m_i)|}{|6-1|} = \frac{1}{6}$.

4.4 Measuring Aspect Cohesion

After measuring the three facets of aspect cohesion independently, we have a discrete view of the cohesion of an aspect. We have two ways to measure the aspect cohesion for an aspect A:

(1) Each measurement works as a field. The aspect cohesion for A is a 3-tuple as $\Gamma(A) = (\gamma_a, \gamma_{ma}, \gamma_m)$.

(2) Integrating the three facets as a whole. Let $x = \beta_1 * \gamma_a + \beta_2 * \gamma_{ma} + \beta_3 * \gamma_m$, the aspect cohesion for A is computed as follows.

$$\Gamma(A) = \begin{cases} 0 & \text{n} = 0 \\ \beta * \gamma_m & \text{k} = 0 \text{ and } n \neq 0 \\ x & \text{others} \end{cases}$$

where k is the number of attributes and n is the number of modules in A, $\beta \in (0, 1]$, $\beta_1, \beta_2, \beta_3 > 0$, and $\beta_1 + \beta_2 + \beta_3 = 1$.

If $k = 0$ and $n \neq 0$, $\Gamma(A)$ describes only the tightness of the call relations, thus we introduce a parameter β to constrain it. For other cases, we introduce three parameters β_1, β_2, and β_3 to constrain it. The selection of β_1, β_2, and β_3 is determined by users.

Example. The aspect cohesion of `PS_Protocol` can be computed based on its γ_a, γ_{ma}, and γ_m. If we set $\beta_1 = \beta_2 = \beta_3 = \frac{1}{3}$, we have $\Gamma(\texttt{PS_Protocol}) = \frac{1}{3} * \gamma_a(\texttt{PS_Protocol}) + \frac{1}{3} * \gamma_{ma}(\texttt{PS_Protocol}) + \frac{1}{3} * \gamma_m(\texttt{PS_Protocol}) = \frac{1}{18}$.

5 Related work

We discuss some related work that directly or indirectly influences our work presented in this paper. To the best of our knowledge, our work is the first attempt to study how to assess the cohesion of aspects in aspect-oriented software.

The approaches taken to measure cohesiveness of procedural programs have generally tried to evaluate cohesion on a procedure (function) by procedure (function) basis. Bieman and Ott [3] propose an approach to measuring the cohesion on procedures based on a relation between output tokens (output variables) and program slices. Kang and Bieman [10] investigate to measure cohesion at the design level for the case that the code has yet to be implemented. Since aspects are more complex and significantly different abstractions in comparing with procedures (functions), These measures definitely fails to be applied to aspects.

Most existing approaches for class cohesion measurement consider the interactions between methods and/or attributes in a class. Chidamber and Kemerer [8] propose the *Lack of Cohesion Measure* (LCOM) to assess class cohesion based on the similarity of two methods in a class. Hitz and Montazeri [9] propose an extension to the LCOM of Chidamber and Kemerer by making it more sensitive

to small changes in the structure of a class. Chae, Kwon, and Bae [6] propose a class cohesion measure for object-oriented system by introducing a new notion called *glue methods*. In contrast to the above cohesion measurement approaches that only consider the interaction between methods and/or attributes, Chen *et al.* [7] propose an approach to measuring class cohesion based on the interactions between attributes and/or methods. Although their work is similar to ours, we see our work differing from theirs because our approach considers more interactions between attributes and modules such as advice, introduction, and pointcuts in an aspect that are unique constructs for aspect-oriented programs. Based on these new types of interactions we propose a new dependence model that are different from the dependence model presented by Chen *et al.* [7].

Zhao [14] proposes a metrics suite for aspect-oriented software, which are specifically designed to quantify the information flows in aspect-oriented programs. To this end, Zhao 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 dependence model, he defines some metrics that can be used to measure the complexity of an aspect-oriented program from various different viewpoints and levels. However, Zhao does not address the issue of aspect cohesion measurement. In addition, Zhao also proposes an approach to assessing the coupling in aspect-oriented systems based on the interactions between classes and aspects [16].

6 Concluding Remarks

In this paper, we proposed an approach to measuring the cohesion of aspects in aspect-oriented software based on dependence analysis. We discussed the tightness of an aspect from three facets: *inter-attribute*, *module-attribute* and *inter-module*. These three facets can be used to measure the aspect cohesion independently and also can be integrated as a whole. We also discussed the properties of these dependencies and according to these properties we proved that our cohesion measures satisfy some properties which a good measure should have. Therefore, we believe our approach may provide a solid foundation for measuring aspect cohesion. In our future work, we will study the influence of aspect inheritance and other aspect-oriented features on aspect cohesion, and apply our cohesion measure approach to real aspect-oriented system design.

Acknowledgments. This work was partially supported by the Japan Society for Promotion of Science (JSPS) under Grand-in-Aid for Scientific Research (C) (No.15500027).

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] J. Bieman and L. Ott. Measuring Functional Cohesion. *IEEE Transactions on Software Engineering*, Vol.22, No.10, pp.644-657, August 1994.

[4] L.C. Briand, J. Daly and J. Wuest. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering*, Vol.3, No.1, pp.65-117, 1998.

[5] L.C. Briand, S. Morasca and V.R. Basili. Defining and Validating Measures for Object-Based High-Level Design. *IEEE Transactions on Software Engineering*, Vol.25, No.5, pp.724-743, 1999.

[6] H.S. Chae, Y. R. Kwon and D. H. Bae. A Cohesion Measure for Object-Oriented Classes. *Software Practice & Experience*, Vol.30, No.12, pp.1405-1431, 2000.

[7] Z. Chen, Y. Zhou, B. Xu, J. Zhao, and H. Yang. A Novel Approach to Measuring Class Cohesion Based on Dependence Analysis. *Proceedings of the International Conference on Software Maintenance*, October 2002.

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

[9] 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.

[10] B. Kang and J. Bieman. Design-Level Cohesion Measures: Derivation, Comparison, and Applications. Computer Science Technical Report CS-96-103, Colarado State University, 1996.

[11] 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.

[12] 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.

[13] 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.

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

[15] J. Zhao and M. Rinard. System Dependence Graph Construction for Aspect-Oriented Programs. Technical Report MIT-LCS-TR-891, Laboratory for Computer Science, MIT, March 2003.

[16] J. Zhao. Coupling Measurement in Aspect-Oriented Systems. Technical Report SE-142-6, Information Processing Society of Japan (IPSJ), June 2003.