3
Aspect-Oriented Software Development

As pointed out in the first chapter, this work is going to tackle the issue of using concern-driven measurement in order to assess the modularity of aspect-oriented design. The goal of this chapter is, therefore, to present the concepts of aspect-oriented software development (Filman et al., 2005). During the rest of this thesis we will refer to the concepts introduced here.

This chapter is structured in three parts. The first part introduces aspect-oriented programming and key abstractions and mechanisms that constitute an aspect-oriented detailed design. The second part concentrates on aspect-oriented software architecture, as our approach targets the assessment of aspect-oriented architectural design. The last part describes existing metrics for conventionally assessing aspect-oriented design modularity. It complements Chapter 2, with a discussion of modularity metrics specific to aspect-oriented design.

3.1.
Aspect-Oriented Programming

Separation of concerns is a fundamental principle that addresses the limitations of human cognition for dealing with complexity. It advocates that to master complexity, one has to deal with one important issue (or concern) at a time (Dijkstra, 1976). In software engineering, the principle of separation of concerns is usually related to system decomposition and modularization (Parnas, 1972): complex software systems should be decomposed into smaller, clearly separated modular units, each dealing with a single concern. The expected benefits are improved comprehensibility and increase on the potential for evolution and reuse in complex software systems.

In software development, the achievement of separation of concerns depends largely on the suitability of abstractions and compositions mechanisms of languages, methods and tools used throughout the software lifecycle. Classes, objects, and methods are examples of classical abstractions in object-oriented

software engineering. For instance, a simple concern can be modularized as a class or as a single method. Inheritance and polymorphism are examples of mechanisms that enable modularization and composition of software concerns.

However, object-orientation has some limitations for dealing with concerns that address requirements involving global constraints and systemic properties, such as synchronization, persistence, error handling, and logging mechanisms, among many others. These concerns have been called crosscutting concerns since they naturally crosscut the boundaries of modular units that implement other concerns. Without proper means for separation and modularization, crosscutting concerns tend to be scattered over a number of modular units and tangled up with other concerns. The natural consequences are lower cohesion and stronger coupling between modular units, reduced comprehensibility, evolvability and reusability of code artifacts.

Aspect-Oriented Programming (AOP) (Kiczales et al., 1997) is an emerging technology that supports a new flavor of separation of concerns at the source code level. It introduces new modularization abstractions and composition mechanisms to improve separation of crosscutting concerns at the implementation level. AOP promotes a new modular unit, called *aspect*, for separating crosscutting concerns and provides new mechanisms for composing aspects with other modular units at well-defined points called *join points*. In the following we briefly describe the main aspect-oriented abstractions and mechanisms. Then we illustrate the use of AOP in the light of an example in AspectJ (The AspectJ Team, 2007; Kickzales et al., 2001), the most well-known AOP language.

Aspects

Aspect is the term used to denote the abstraction that aims to support improved isolation of crosscutting concerns. Aspects are modular units of crosscutting concerns that are associated with a set of classes or objects. An aspect can affect, or crosscut, one or more classes and/or objects in different ways. Aspect-oriented system designs are decomposed into classes and aspects; aspects modularize crosscutting concerns and classes modularize non-crosscutting concerns. In addition to conventional attributes and methods, an aspect includes pointcuts and pieces of advice as described bellow.

Join Points and Pointcuts

Essential to the process of composing aspects and classes is the concept of *join points*, the elements that specify how classes and aspects are related. Join points are well-defined points in the dynamic execution of a system. Examples of join points are method calls, method executions, attributes sets and reads, and object initialization. Each aspect defines one or more first-order logic expressions, called *pointcut expressions* (or just pointcuts), to select the join points that will be affected by the aspect's crosscutting behavior.

Advice

When program execution reaches a join point selected by some pointcut expression, a body of code, called *advice*, can be executed before, after or around it. Advice is a special method-like construct attached to pointcuts. There are different kinds of advice: (i) a before advice runs whenever a join point is reached and before the actual computation proceeds, (ii) an after advice runs after the computation under the join point finishes, i.e. after the method body has run, and just before control is returned to the caller, and (iii) an around advice runs whenever a join point is reached, and has explicit control whether and when the computation under the join point is allowed to run at all.

Currently, AspectJ (The AspectJ Team, 2007; Kickzales et al., 2001) is the most well-known general-purpose language for AOP. It is an extension to the Java programming language. The aforementioned concepts – aspects, pointcuts, join points, advice – constitute a common standard vocabulary for AOP adopted from AspectJ (The AspectJ Team, 2007). Additionally, aspects in AspectJ can provide intertype declarations, which are attributes and methods that will be inserted into classes.

Figure 3 shows an example of an aspect obtained in the AspectJ Programming Guide (The AspectJ Team, 2007). The FaultHandler aspect consists of an inter-type declaration which introduces an attribute in the Server class (line 03), two conventional methods (lines 05-07 and 08-10), a pointcut (line 12) and two pieces of advice (lines 14-16 and 17-20). This covers the basics of what aspects can contain. The pointcut, named services, defines as join points the call to any public method of objects of type Server. This is specified by the clausule call(public * * (..)). It also allows any piece of advice using the services pointcut to

access the Server object whose method is being called. This is specified by the clausule target(s).

The piece of advice in lines 14-16 specifies that the piece of code in line 15 is executed when instances of the Server class have their public methods called, as specified by the pointcut services. More specifically, it runs when those calls are made, just before the corresponding methods are executed. The piece of advice in lines 17-20 defines another piece of code that is also executed on the services pointcut. However, in this case, the piece of code is executed after the called method throw exception of type FaultException.

```
01 aspect FaultHandler {
02
03   private boolean Server.disabled = false;
04
05   private void reportFault() {
06     System.out.println("Failure! Please fix it!.");
07   }
08   public static void fixServer(Server s) {
09     s.disabled = false;
10   }
11
12   pointcut services(Server s): target(s) && call(public * * (..));
13
14   before(Server s): services(s) {
15     if (s.disabled) throw new DisabledException();
16   }
17   after(Server s) throwing (FaultException e): services(s) {
18     s.disabled = true;
19     reportFault();
20   }
21 }
```

Figure 3: Example of an aspect in AspectJ

Figure 4 presents a didactic example that shows the difference between Java and AspectJ implementations of the same program (The AspectJ Team, 2007). It shows the code of a simple program to manage graphical elements. The Java solution (left side of Figure 4) encompasses the classes Point, Line and Display (the latter is not shown in the figure). The AspectJ implementation (right side of Figure 4) comprises the same classes plus the DisplayUpdating aspect. This example shows that the method update of the class Display must be called after every call to methods setX and setY of the class Point and methods setP1 and setP2 of the class Line. In the Java implementation, the call to Display.update is spread over the four methods since it is explicitly done at the end of each of them (lines 9, 13, 25 and 19). In the AspectJ solution, this call is localized only in the

DisplayUpdating aspect (line 37) and is executed when the join points defined by the move pointcut (lines 30-34) are reached.

```
01 class Point {
02  private int x = 0, y = 0;
03
04  int getX() { return x; }
05  int getY() { return y; }
06
07  void setX(int x) {
08    this.x = x;
09    Display.update();
10  }
11  void setY(int y) {
12    this.y = y;
13    Display.update();
14  }
15 }
16
17 class Line {
18  private Point p1, p2;
19
20  Point getP1() { return p1; }
21  Point getP2() { return p2; }
22
23  void setP1(Point p1) {
24    this.p1 = p1;
25    Display.update();
26  }
27  void setP2(Point p2) {
28    this.p2 = p2;
29    Display.update();
30  }
31 }
```

```
01 class Point {
02  private int x = 0, y = 0;
03
04  int getX() { return x; }
05  int getY() { return y; }
06
07  void setX(int x) {
08    this.x = x;
09  }
10  void setY(int y) {
11    this.y = y;
12  }
13 }
14
15 class Line {
16  private Point p1, p2;
17
18  Point getP1() { return p1; }
19  Point getP2() { return p2; }
20
21  void setP1(Point p1) {
22    this.p1 = p1;
23  }
24  void setP2(Point p2) {
25    this.p2 = p2;
26  }
27 }
28
29 aspect DisplayUpdating {
30  pointcut move():
31    call(void Line.setP1(Point))||
32    call(void Line.setP2(Point))||
33    call(void Point.setX(int))  ||
34    call(void Point.setY(int));
35
36  after() returning: move() {
37    Display.update();
38  }
39 }
```

Figure 4: Java (left side) and AspectJ (right side) version of the same program.

3.2. Aspect-Oriented Architecture Design

Aspect-oriented abstractions and related composition mechanisms have been also discussed with the goal of supporting the separation of crosscutting concerns in other phases of the software life cycle. In the context of software architecture, a number of aspect-oriented architecture description languages have been proposed to allow the representation of aspect-oriented abstractions at the architectural design level. Architecture description languages (ADLs) are modeling notations to support architecture-based development (Medvidovic & Taylor, 2000). An ADL focuses on the high-level structure of the overall software

rather than the implementation details of any specific source component (Medvidovic & Taylor, 2000).

DAOP-ADL (Pinto et al., 2003), Fractal ADL (Pessemier et al., 2004), AO-ADL (Pinto & Fuentes, 2007) and AspectualACME (Garcia et al., 2006a) are examples of aspect-oriented ADLs. Moreover, some graphical notations, such AOGA (Garcia, 2004; Kulesza et al., 2004) and AO Visual Notation (Tekinerdoğan et al., 2006) provide graphical notations for modeling aspect-oriented component-and-connector (C&C) views (Bass et al., 2003). This section describes existing approaches for specifying aspect-oriented architectures. Some of these approaches are later used in our empirical studies (Chapter 7), namely AOGA and AO Visual Notation, and supported by our measurement tool (Section 6.2), namely AO-ADL.

A C&C view (Bass et al., 2003) is an architecture view in which the elements are components and connectors. Components are main units of computation; connectors are the communication means between components. Components and the connectors are attached to each other. C&C views consist of the major executing components and how they interact. Architectural aspects (or aspectual components) can be defined both to modularize architectural crosscutting concerns and to separate them from other architectural components. Architectural aspects may affect components at well-defined architectural join points. For instance, an architectural join point can be the invocation of an operation of some component interface.

Most aspect-oriented ADLs are motivated by the integration of existing ADL concepts (e.g. component, interface, and connector) with new AO abstractions (e.g. aspect, join point, pointcut and advice) in order to address the modeling of crosscutting concerns in architecture. Navasa et al (2002) define a set of requirements which current ADLs need to address to allow the management of crosscutting concerns using architectural connection abstractions. The requirements are: (i) definition of primitives to specify join points in functional components, (ii) definition of the aspect abstraction as a special kind of component, and (iii) specification of connectors between joinpoints and aspects. The authors suggest the use of existing coordination models to specify the connectors between functional components and aspects.

Pinto et al (2003) propose DAOP-ADL which considers components and aspects as first-order elements. Aspects can affect the components' interfaces by means of: (i) an evaluated interface which defines the operations that aspects are able to affect, and (ii) a target event interface responsible for describing the events that an aspect can capture. The connection between components and aspects is supported by a set of aspect evaluation rules. They define when and how the aspect behavior is executed.

Pessemier et al (2004) extend the Fractal ADL with *aspect components*. Aspect components are responsible for specifying existing crosscutting concerns in the software architecture. Each aspect component can affect components by means of a special interception interface. Two kinds of connections between components and aspect components are offered: (i) a direct crosscut connection by declaring the component references, and (ii) a crosscut connection using pointcut expressions based on component names, interface names and service names.

AspectualACME (Garcia et al., 2006a) is a simple and seamless extension of the ACME ADL (Garlan et al., 1997) to support the modular representation of architectural aspects and their multiple composition forms. AspectualACME promotes a natural blending of aspects and architectural abstractions by employing a special kind of architectural connector, called Aspectual Connector, to encapsulate aspect-component connection details.

Pinto & Fuentes (2007) proposed a XML-based aspect-oriented ADL called AO-ADL. The structural organization of AO-ADL is based on the fact that the main difference of architectural crosscutting and non-crosscutting concerns is in the role they play in a particular composition binding and not in the internal behavior itself. Therefore, differently from the previously mentioned ADLs, AO-ADL does not include a new element to model aspects. Components in AO-ADL model either crosscutting or non-crosscutting behavior. This is called a symmetric approach. Thus, a component is considered an aspect when it participates in an aspectual interaction. In this context, another contribution of AO-ADL is the extension of the semantic of conventional connectors to represent the crosscutting effect of "aspectual" components. This means that AO-ADL connectors provide support to describe not only typical communication as in traditional ADLs, but also crosscutting influence among components.

The aforementioned aspect-oriented ADLs can be classified in two categories: symmetric and asymmetric approaches. The asymmetric ADLs, such as DAOP-ADL (Pinto et al., 2003) and Fractal ADL (Pessemier et al., 2004), include a special abstraction to represent "aspectual components" at the architecture description. Aspectual components comprise special kinds of interfaces to specify the points at the architecture affected by it. We could say that the pointcuts are specified in the interfaces, such as the DAOP-ADL "evaluated interface".

On the other hand, symmetric aspect-oriented ADLs, such as AO-ADL (Pinto & Fuentes, 2007) and AspectualACME (Garcia et al., 2006a) do not define any special kind of component to model aspects. Instead, crosscutting and non-crosscutting concerns are captured by conventional components without any special kind of interface. These ADLs rely on a special kind of connector to represent crosscutting relationship between components playing the role of aspects and the other components. In this case, we could say that the pointcuts are specified in the connectors.

In our empirical studies (Chapter 7), we used graphical notations to represent the architectures under assessment. In particular, we used an asymmetric notation provided by AOGA (Garcia, 2004; Kulesza et al., 2004) in the first study and a symmetric notation, called AO Visual Notation (Tekinerdoğan et al., 2006), in the last two studies. In the following sections, we describe these notations.

3.2.1. AOGA

AOGA is a graphical notation for describing C&C views of aspect-oriented architectures. In AOGA, the architect has modeling support to distinguish between normal components and aspectual components. Aspectual components are aspects at the architectural level. An aspectual component is represented like a UML 2.0 component (OMG, 2005) with a diamond on the top of it, as shown in Figure 5. Each aspectual component can be related to more than one conventional or aspectual component, representing its crosscutting nature.

In AOGA models, interfaces are attached to the architectural components. The interfaces are categorized in three groups: provided interfaces, required

interfaces and crosscutting interfaces. Figure 5 illustrates AOGA notation for architectural components and interfaces. Provided and required interfaces are represented as defined in UML 2.0 (OMG, 2005). Crosscutting interfaces are represented as small gray circles. Each interface has a name, which is placed next to it. Each architectural component has one or more interfaces.

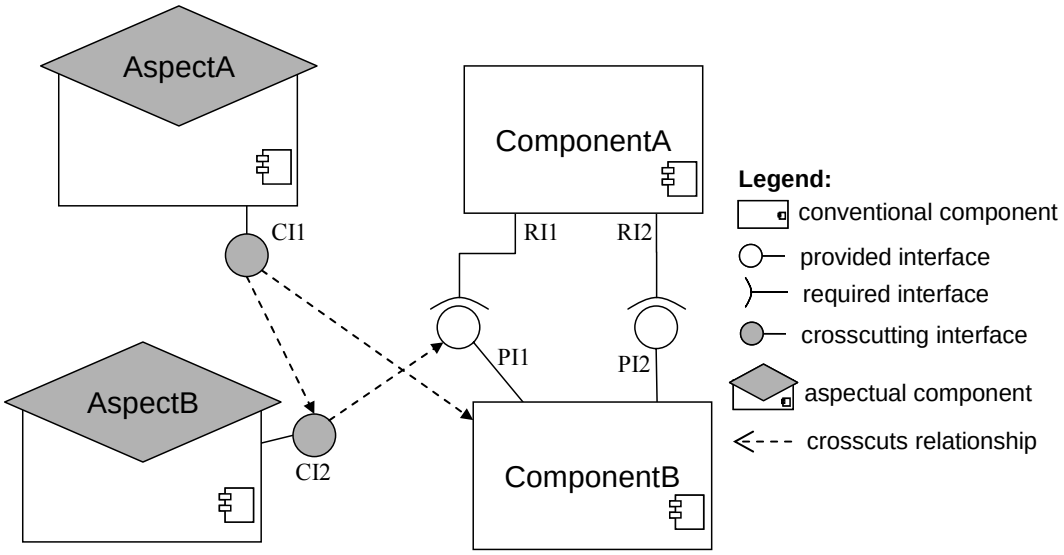

Figure 5: AOGA architecture elements

Crosscutting interfaces specify which architectural components are affected by aspectual components. However, they do not declare how the components are affected. A crosscutting interface is different from a provided interface. The latter only provides services to other components. Besides providing services, crosscutting interfaces also specify when an architectural aspect affects other architectural components. An aspectual component conforms to a set of crosscutting interfaces. The operations declared in an interface represent the services provided by it.

A crosscutting interface can be attached to either internal elements of the architectural components or other interfaces by means of *crosscuts* relationships. The first case means that the architectural aspect directly affects the internal structure or dynamic behavior of the target component. The second case means that the aspectual component affects the operations defined by other interfaces.

3.2.2.
AO Visual Notation

The AO Visual Notation (Tekinerdoğan et al., 2006) extends the set of architecturally-relevant abstractions and respective graphical elements of UML 2.0 (OMG, 2005), such as components, interfaces, and connectors. AO Visual Notation is a symmetric approach, thus, both crosscutting and non-crosscutting concerns are represented by components. The distinction is made at the connector level.

The AO Visual Notation provides support for architecture-level crosscutting compositions by means of the notion of *aspectual connectors*. The authors claim that conventional connector types, available in UML 2.0, are not appropriate to capture the notion of crosscutting compositions. The reason is that conventional connectors must only be defined from a required interface to a provided interface. This rule violates a typical composition property of crosscutting collaborations, which specifies that an aspectual component and affected components can be linked through their both provided interfaces (Garcia et al., 2006a; Kulesza et al., 2004).

Figure 6 shows how aspectual connectors are represented in the AO Visual Notation. The use of the stereotype is optional. The aspectual connector is a component-like graphical notation with elements to specify the "crosscutting collaboration" between the involved architectural elements.

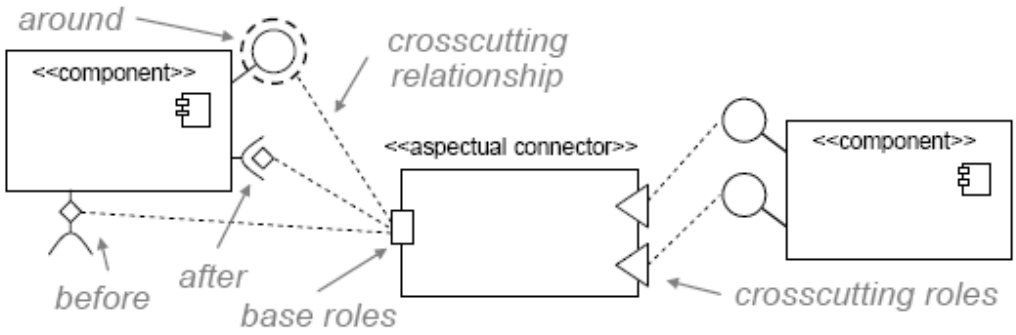

Figure 6: AO Visual Notation: Aspectual Connectors

Aspectual connectors (Figure 6) are basically formed by base and crosscutting roles. These roles consist of two types of connector's interfaces, and define the role the connected components are playing in a crosscutting

composition. A crosscutting role defines which component is playing the role of "aspect" in the architecture decomposition. Crosscutting roles are represented by triangles "cutting across" the connector boundaries. Base roles are associated with different join points affected by the components associated with the crosscutting roles. They are represented by small rectangles in the opposite extreme of an aspectual connector (Figure 6).

Crosscutting relationships define how the connectors and components are attached. They are equivalent to attachments in conventional ADLs, such as ACME (Garlan et al., 1997). Their graphical representation is a dashed line. The dashed lines associate crosscutting or base roles with component interfaces. The set of join points of interest in a certain crosscutting composition are conventionally indicated by visual (and sometimes, textual) elements associated with a crosscutting relationship. The three interfaces of the component on the left of Figure 6 are associated with crosscutting relationships. These three interfaces are join points affected by the component on the right of the figure, which plays the role of an aspect.

When a component interface is touched by a line, it means that one or more of the interface operations are affected by an aspectual connector. Whenever it is required, a sequencing operator can be associated with a crosscutting relationship. It specifies when or how the connector is affecting the operation(s). The notation includes graphical elements for three sequencing operators: before, after, and around (Figure 6). For the sake of scalability, a simpler notation for aspectual connectors is available in case connector internals are not relevant (Figure 7).

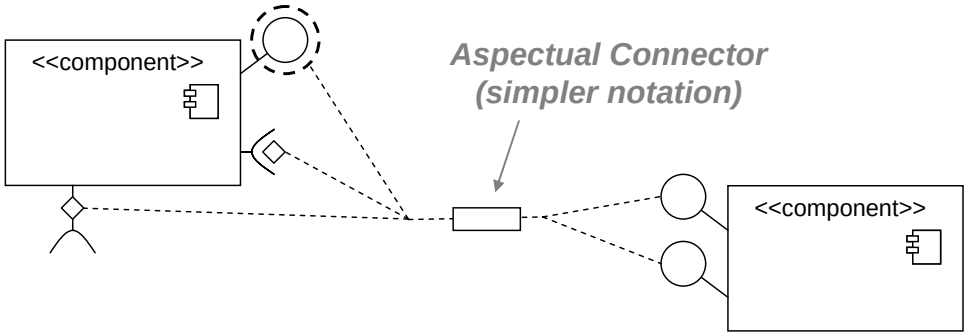

Figure 7: Simpler notation for aspectual connectors

3.3. Aspect-Oriented Metrics

Aspect-oriented software development and AspectJ, in particular, have received an increasing amount of attention by the measurement community. Metrics tailored to be applied to aspect-oriented abstractions have been proposed. In fact, most of these metrics rest upon AspectJ mechanisms. While object-oriented metrics are defined in terms of classes, methods and attributes, aspect-oriented metrics have been defined in terms of aspects, advice, pointcuts and intertype declarations. Aspect-oriented programs also include classes, methods and attributes, thus the definition of aspect-oriented metrics also consider these abstractions.

Most of the aspect-oriented metrics for quantifying modularity-related attributes, such as coupling and cohesion, are extensions of existing object-oriented metrics (Ceccato & Tonella, 2004; Sant'Anna et al., 2003; Zhao, 2002, 2004; Zhao & Xu, 2004). These metrics can be considered as conventional as object-oriented metrics in the sense that they are defined upon module-like abstractions, such as classes and aspects. Therefore, existing aspect-oriented metrics suffer from the same limitations of existing object-oriented metrics – they are not calibrated by the system's concerns (Section 1.2). In the following subsections, we briefly describe three relevant representative suites of aspect-oriented metrics, in order to show the new dimensions of conventional modularity measurement that are imposed by aspect-oriented design. The last subsection (Section 3.3.4) discusses a new type of connection considered by aspect-oriented coupling metrics.

3.3.1. Metrics by Ceccato & Tonella

Ceccato & Tonella (2004) define five coupling metrics for aspect-oriented software: Coupling on Advice Execution (CAE), Coupling on Intercepted Modules (CIM), Coupling on Method Call (CMC), Coupling on Field Access (CFA), and Crosscutting Degree of an Aspect (CDA). CAE is defined as the number of aspects containing advice that is possibly triggered by the execution of operations in a given class or aspect. CIM is defined as the number of classes or

aspects explicitly named in the pointcuts of a given aspect. CMC is defined as the number of classes or aspects declaring methods that are possibly called by a given class or aspect. CFA is defined as the number of classes or aspects declaring fields that are accessed by a given class or aspect. CDA is defined as the number of classes affected by the pointcuts and inter-type declarations of a given aspect.

Ceccato & Tonella (2004) define one cohesion metric: Lack of Cohesion in Operations (LCO). This metric is a direct adaptation of the Chidamber & Kemerer's LCOM metric (Section 2.4). It is defined as the number of pairs of operations (methods or pieces of advice) working on different class fields minus pairs of operations working on common fields. Response for a Module (RFM) is another metrics defined by Ceccato & Tonella. This metric is an adaptation of Chidamber & Kemerer's RFC metric (Section 2.4). In addition, the RFM now also includes aspects and take into account the pieces of advice that might be executed due to pointcuts.

3.3.2. Metrics by Sant'Anna et al.

Sant'Anna et al. (2003) define coupling and cohesion metrics for aspect-oriented software[1]. Coupling between Components (CBC) is an extension of Chidamber & Kemerer's Coupling Between Object Classes (CBO) metric (Section 2.4). A component is defined as a class or an aspect. Thus, CBC is defined for a class or an aspect as the number of other classes or aspects to which it is coupled. Their definition mentions pointcuts as one of the considered coupling dimension between classes and aspects. They proposed the Lack of Cohesion in Operations (LCOO) metric. It measures the amount of advice/method pairs that do not access the same instance variable and is thus an extension of the LCOM metric by Chidamber & Kemerer (Section 2.4).

[1] These metrics are not contribution of this thesis. They were proposed in the context of Sant'Anna's master dissertation.

3.3.3. Metrics by Zhao and Xu

Zhao and Xu's metrics (Zhao, 2002, 2004; Zhao & Xu, 2004) are based on a dependence model for aspect-oriented software that consists of a group of dependence graphs. The coupling metrics are: Attribute-Class Dependence, Module-Class (member-class) Dependence, Module-Method (member-method) Dependence, and Aspect-Inheritance Dependence. The Attribute-Class Dependence metric relates to the dependence between attributes of an aspect and classes. The Module-Class (member-class) Dependence measure relates to the dependence between members of an aspect and classes. According to their definition of members of an aspect, this measure can be subdivided into advice-class, intertype-class, method-class and the pointcut-class dependence measure. The Module-Method (member-method) Dependence measure relates to the dependence between members of an aspect and methods of a class and can be subdivided into four dimensions: advice-method, intertype-method, method-method and pointcut-method dependence.

3.3.4. Connection between Aspects and Classes

In Section 2.4, we listed the possible types of connections in object-oriented coupling metrics identified by Briand et al. (1999). The aspect-oriented metrics also take into account all those types of connections, since an aspect-oriented program consists of classes and aspects. However, new types of connections are needed to compute aspect-oriented metrics. Based on the aforementioned suites of aspect-oriented metrics, we can observe that there is a very relevant new type of connection between aspects and classes which could be described as "an aspect *a* affects a class *c* by means of a pointcut". At the architecture design we could define this coupling dimension as "an aspectual component *a* affects a component *c* by means of a crosscutting relationship". Our suite of architectural metrics also includes coupling metrics. Our assessment approach also targets aspect-oriented design, thus our coupling between component metrics (Section 4.3.6) take into account this new type of connection between aspectual components and conventional components.