Measuring the Effects of Software Aspectization

Mariano Ceccato and Paolo Tonella
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
{ceccato, tonella}@itc.it

Abstract

The aim of Aspect Oriented Programming (AOP) is the production of code that is easier to understand and evolve, thanks to the separation of the crosscutting concerns from the principal decomposition. However, AOP languages introduce an implicit coupling between the aspects and the modules in the principal decomposition, in that the latter may be unaware of the presence of aspects that intercept their execution and/or modify their structure. These invisible connections represent the main drawback of AOP. A measuring method is proposed to investigate the trade-off between advantages and disadvantages obtained by using the AOP approach. The method that we are currently studying is based on a metrics suite that extends the metrics traditionally used with the OO paradigm.

1 Introduction

When existing software is migrated to Aspect Oriented Programming (AOP), crosscutting concerns are separated from the principal decomposition and are encapsulated inside dedicated modularization units (aspects). Maintenance of the resulting code is expected to be easier, thanks to the possibility of modifying locally the crosscutting behavior. However, a novel kind of (implicit) coupling is introduced by AOP languages. In fact, the code that belongs to the principal decomposition might be unaware of the presence of aspects that intercept its execution and/or modify its structure. This creates a twofold dependence: on one hand, the aspect code works properly only under given assumptions on the code in the principal decomposition. Such assumptions may become invalid during code evolution. On the other hand, the overall behavior depends both on the code in the principal decomposition *and* on the aspect code, so that a change in the latter might affect the former. If not controlled, such kind of coupling might reduce or cancel at all the potential benefits coming from the separation of crosscutting functionalities from the principal decomposition.

The position of the authors is that the trade-off between the advantages obtained from the separation of concerns and the disadvantages caused by the coupling introduced by the aspects must be investigated in more detail, in order for AOP to gain a wider acceptance. Empirical studies should be conducted to evaluate costs and benefits offered by the AOP solution with respect to the more traditional, Object-Oriented (OO) one, in terms of code understandability, evolvability, modularity and testability. Moreover, alternative AOP solutions could be contrasted empirically, in order to identify good/bad AOP practices, to be possibly encoded into a catalog of AOP patterns/anti-patterns.

The first step in this direction is the definition of a set of metrics to quantitatively assess the effects of the software "aspectization". Such metrics can be based on those widely used with OO software. Although some extensions of OO metrics to AOP are available in the literature [**?**, **?**, **?**, **?**, **?**, **?**], none seems to address explicitly all the different kinds of coupling that aspects and objects can have between each other.

In the remaining of this paper we discuss OO metrics (Sec. 2) and consider their extension to AOP (Sec. 3). Then, our AOP metrics tool is described (Sec. 4), followed by its usage on an example (Sec. 5). Related works (Sec. 6) and conclusions (Sec. 7) terminate the paper.

2 OO metrics

The inadequacy of the metrics in use with procedural code (size, complexity, etc.), when applied to OO systems, led to the investigation and definition of several metrics suites accounting for the specific features of OO software. However, among the available proposals, the one that is most commonly adopted and referenced is that by Chidamber and Kemerer [2]. We argue that a shift similar to

the one leading to the Chidamber and Kemerer's metrics is necessary when moving from OO to AOP software.

Some notions used in the Chidamber and Kemerer's suite can be easily adapted to AOP software, by unifying classes and aspects, as well as methods and advices. Aspect introductions and static crosscutting require minor adaptations. However, novel kinds of coupling are introduced by AOP, demanding for specific measurements. For example, the possibility that a method execution is intercepted by an aspect pointcut, triggering the execution of an advice, makes the intercepted method coupled with the advice, in that its behavior is possibly altered by the advice. In the reverse direction, the aspect is affecting the module containing the intercepted operation, thus it depends on its internal properties (method names, control flow, etc.) in order to successfully redirect the operation's execution and produce the desired effects.

In the following section, the Chidamber and Kemerer's metrics suite is revised. Some of the metrics are adapted or extended, in order to make them applicable to the AOP software.

3 AOP metrics

Since the proposed metrics apply both to classes and aspects, in the following the term *module* will be used to indicate either of the two modularization units. Similarly, the term *operation* subsumes class methods and aspect advices/introductions.

WOM (Weighted Operations in Module): *Number of operations in a given module.*

Similarly to the related OO metric, WOM captures the internal complexity of a module in terms of the number of implemented functions. A more refined version of this metric can be obtained by giving different weights to operations with different internal complexity.

DIT (Depth of Inheritance Tree): *Length of the longest path from a given module to the class/aspect hierarchy root.*

Similarly to the related OO metric, DIT measures the scope of the properties. The deeper a class/aspect is in the hierarchy, the greater the number of operations it might inherit, thus making it more complex to understand and change. Since aspects can alter the inheritance relationship by means of static crosscutting, such effects of aspectization must be taken into account when computing this metric.

NOC (Number Of Children): *Number of immediate subclasses or sub-aspects of a given module.*

Similarly to DIT, NOC measures the scope of the properties, but in the reverse direction with respect to DIT. The number of children of a module indicates the proportion of modules potentially dependent on properties inherited from the given one.

CAE (Coupling on Advice Execution): *Number of aspects containing advices possibly triggered by the execution of operations in a given module.*

If the behavior of an operation can be altered by an aspect advice, due to a pointcut intercepting it, there is an (implicit) dependence of the operation from the advice. Thus, the given module is coupled with the aspect containing the advice and a change of the latter might impact the former. Such kind of coupling is absent in OO systems.

CIM (Coupling on Intercepted Modules): *Number of modules or interfaces explicitly named in the pointcuts belonging to a given aspect.*

This metric is the dual of CAE, being focused on the aspect that intercepts the operations of another module. However, CIM takes into account only those modules and interfaces an aspect is aware of – those that are explicitly mentioned in the pointcuts. Sub-modules, modules implementing named interfaces or modules referenced through wildcards are not counted in this metric, while they are in the metric CDA (see below), the rationale being that CIM (differently from CDA) captures the *direct* knowledge an aspect has of the rest of the system. High values of CIM indicate high coupling of the aspect with the given application and low generality/reusability.

CMC (Coupling on Method Call): *Number of modules or interfaces declaring methods that are possibly called by a given module.*

This metric descends from the OO metric CBO (Coupling Between Objects), which was split into two (CMC and CFA) to distinguish coupling on operations from coupling on attributes. Aspect introductions must be taken into account when the possibly invoked methods are determined. Usage of a high number of methods from many different modules indicates that the function of the given module cannot be easily isolated from the others. High coupling is associated with a high dependence from the functions in other modules.

CFA (Coupling on Field Access): *Number of modules or interfaces declaring fields that are accessed by a given module.*

Similarly to CMC, CFA measures the dependences of a given module on other modules, but in terms of accessed

fields, instead of methods. In OO systems this metric is usually close to zero, but in AOP, aspects might access class fields to perform their function, so observing the new value in aspectized software may be important to assess the coupling of an aspect with other classes/aspects.

RFM (Response For a Module): *Methods and advices potentially executed in response to a message received by a given module.*

Similarly to the related OO metric, RFM measures the potential communication between the given module and the other ones. The main adaptation necessary to apply it to AOP software is associated with the *implicit* responses that are triggered whenever a pointcut intercepts an operation of the given module.

LCO (Lack of Cohesion in Operations): *Pairs of operations working on different class fields minus pairs of operations working on common fields (zero if negative).*

Similarly to the LCOM (Lack of Cohesion in Methods) OO metric, LCO is associated with the pairwise dissimilarity between different operations belonging to the same module. Operations working on separate subsets of the module fields are considered dissimilar and contribute to the increase of the metric's value. LCO will be low if all operations in a class or an aspect share a common data structure being manipulated or accessed.

CDA (Crosscutting Degree of an Aspect): *Number of modules affected by the pointcuts and by the introductions in a given aspect.*

This is a brand new metric, specific to AOP software, that must be introduced as a completion of the CIM metric. While CIM considers only explicitly named modules, CDA measures all modules possibly affected by an aspect. This gives an idea of the overall impact an aspect has on the other modules. Moreover, the difference between CDA and CIM gives the number of modules that are affected by an aspect without being referenced explicitly by the aspect, which might indicate the degree of generality of an aspect, in terms of its independence from specific classes/aspects. High values of CDA and low values of CIM are usually desirable.

The proposed metric suite has no completeness claim and needs to be adapted for specific measurement goals (e.g., following the GQM approach [1]). While all the proposed metrics can be used to compare alternative AOP implementations, not all of them can be applied when an OOP program is migrated to AOP. CAE and CIM do not make sense in OOP, thus an overall TC (Total Coupling) metric should be used instead, counting the total number of coupling relationships between modules (either of type CAE, CIM, CMC or CFA). Of course, this is not the sum of the four metrics. Individual coupling metrics are still of interest to understand where a given TC value originates from. Similarly to CAE and CIM, CDA does not apply to OOP. However, its value for the migrated AOP program is interesting when compared to CIM, as explained above.

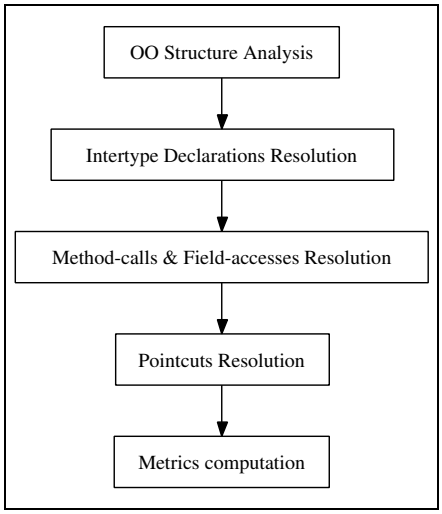

Figure 1. *Metrics tool.*

4 Metrics tool

To assess the proposed metrics suite, we developed an AOP metrics tool that computes all the proposed measures for code written in the AspectJ [5] language. The tool exploits a static analyzer developed in TXL [3]. Figure 1 shows the internal organization of the tool, focusing on the modules required to compute the AOP metric values.

The first module of the tool takes as input all the source classes, interfaces and aspects and performs some standard static OO code analysis, to detect the structure of the modules, in terms of their fields, operations and inheritance relationships. Such information is stored in a data base.

After the first module, the second one can be run, performing more accurate analysis. Each aspect is processed for a second time in order to detect the inter-type declarations, in terms of field introductions, method introductions and changes of class/interface inheritance relationships. The resulting data are stored in the same data base, being associated to the target class as if the information came from the analysis of the class itself. The name of the aspect responsible for such introductions is however recorded. In this way the first step of the weaving algorithm is realized.

The next module of the tool detects the method-call relationship. Moreover, it discovers the field-access relationship between operations and fields (both belonging to the same module or to other modules). For such an analysis a symbol table, mapping the variables to the respective type, is maintained and pushed onto the stack whenever a new scope is opened. The symbol table is required to know the type of each method invocation target, return value and accessed field. Such type information is stored in the database under construction. Polymorphic calls are resolved conservatively with all methods that possibly override the invoked one.

The fourth step is the most complex one. It completes weaving by resolving all the pointcuts in the aspect code, thus producing the corresponding join points in the intercepted code. The algorithm for this phase starts coping with the primitive pointcuts, which are resolved using the inheritance, invocation and access information collected so far. Then, it composes the join points according to the union, intersection and negation operators used in the pointcut specifications. When all the pointcuts are resolved, the advices can be associated to the advised methods, storing this relationship in the available data base

The final step concerns the computation of the metrics. The value of a metric for a given module is obtained just by running a query on the database. The overall value of a metric for the whole system is computed as the median of the values computed for all the modules in the system.

5 Example

The proposed metrics have been computed for a small example, taken from the implementation of some design patterns [4] provided by Jan Hannemann[1] both in Java and in AspectJ.

Our test is the implementation of the *Observer* design pattern [4], in which there are two distinct roles, the *Subject* and the *Observer*. The Subject is an entity that can be in several different states. Some of the state changes are of interest to the Observer, which may take some actions in response to the change.

The Observer pattern requires that the Observer registers itself on those Subjects it intends to observe. The Subject maintains a list of the Observers registered so far. When the Subject changes its state, it notifies the Observers of the change, so that the Observers can take the appropriate actions.

In the OO implementation by Jan Hannemann, this design pattern consists of two interfaces, `ChangeSubject` and `ChangeObserver`, with the abstract definitions of the Subject and Observer roles. Moreover, the implementation contains the `Point` and the `Screen` classes, the first playing the role of Subject whereas the second plays both roles in two different instances of the pattern. The `Main` class contains the code to set up the two different pattern instances and run them. In the first pattern instance `Point` acts as the Subject and `Screen` as the Observer. In the second case, an instance of the class `Screen` is the Subject, while other instances of the same class are its Observers.

The AOP implementation contains a different version of the classes `Point` and `Screen`, with no code regarding the Subject/Observer roles. `ObserverProtocol` is an abstract aspect defining the general structure of the aspects that implement the Observer pattern. This abstract aspect is extended by `ScreenObserver`, `ColorObserver` and `CoordinateObserver`. These concrete aspects contain the actual implementation of the protocol. By means of inter-type declarations, they impose roles onto the involved classes and by means of appropriate pointcuts they specify the Subject actions to be observed. Moreover, these aspects contain the mapping that connects a Subject to its Observers. The class `Main` runs the code for the initialization of the patterns an for their execution.

version	WOM	DIT	NOC	*TC*	RFM	LCO	CDA
java	3	1	0	*2*	7	1-12	N.A.
aspectj	1	2	0	*3*	2	0	3

Table 1. *Metrics for the Observer design pattern.*

version	CAE	CIM	CMC	CFA	TC
java	0	0	2	0	**2**
aspectj	0	2	1	0	**3**

Table 2. *Coupling Metrics for the Observer design pattern.*

We applied our metric suite to the two implementations of the Observer pattern. The median values produced by the tool are shown in Table 1. The value of LCO for the OO code is indicated as 1-12, since these two values are adjacent to the median point. The *TC* column contains the value for total coupling. Detailed values for all different coupling kinds are shown in Table 2.

We can notice a general improvement of some metrics (WOM, LCO, and RFM), no change a metric (NOC) and a worse value of DIT (due to the super-aspect `ObserverProtocol`) and of TC. While in general the values change only a little bit, for RFM the change is relatively high, passing from 7 to 2. LCO is also affected positively, going from 1-12 to 0. The cost to be paid for

[1] http://www.cs.ubc.ca/~jan/AODPs

such improvements is an increase of the coupling metric TC as expected. Looking at Table 2, we can have a detailed insight on the reasons for the coupling increase. Even if there is a decrease of the method coupling (CMC) there is a much bigger increasing of the coupling regarding the aspects which intercept method executions (CIM). However, the fact that the value of CAE is higher than that of CIM indicates that the aspects have only a partial knowledge of the classes they are affecting and contain quite generic, independent pointcut definitions.

6 Related work

The cohesion measure called *Module-Attribute Cohesion* in [**?**] is based on the same dependences between operations and fields that we consider in our LCO metric, but, differently from our metric, it is not an extension of the LCOM metric proposed in [2]. As regards the proposed coupling metrics, while CIM, CMC and CAE correspond to the *Pointcut-class*, *Method-method* and *Pointcut-method* dependence measures presented in [**?**], CDA has no counterpart in [**?**].

Similarly to us, the authors of [**?**, **?**] considered the Chidamber and Kemerer's metric suite, properly adapted to AOP. However, they do not recognize the different nature of the various kinds of coupling introduced by the aspects. The authors of [**?**] added a few metrics to capture the level of scattering of the application concerns. However, the definition of such metrics (*SoC metrics*) is not operational, thus making it difficult to compute them automatically. The expected effects of AOP on the Chidamber and Kemerer's metrics are analyzed in [**?**].

The indications in [**?**, **?**] on the definition of cohesion and coupling metrics for OO systems will be considered in our future work, in order to possibly refine the proposed AOP metrics for such attributes.

7 Conclusions

Most research in AOP is focused on new design processes, languages and frameworks to support the new paradigm. However, no strong empirical evaluation was conducted to assess the effects of AOP adoption. The first step in this direction consists of defining a metrics suite for AOP software, designed so as to capture the novel features introduced by this programming style. We contributed to the ongoing discussion on such metrics by distinguishing among the different kinds of coupling relationships that may exist between modules and by proposing a new metric for the crosscutting degree of an aspect (CDA). Moreover, we conducted a small case study to evaluate the information carried by the proposed metrics when applied to an OO system and to the same system migrated to AOP. Results indicate that meaningful properties, such as the proportion of the system impacted by an aspect and the amount of knowledge an aspect has of the modules it crosscuts, are captured by the proposed metrics (CDA and CIM repsectively). We envisage the definition of a common set of AOP metrics, to be adopted by the AOP community, in order to simplify the comparison of the results obtained by different research teams and to have a standard evaluation method.

References

[1] V. Basili, G. Caldiera, and D. H. Rombach. *The Goal Question Metric Paradigm, Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[2] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[3] J. Cordy, T. Dean, A. Malton, and K. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13):827–837, 2002.

[4] E. Gamma, R. Helm, R.Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley Publishing Company, Reading, MA, 1995.

[5] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, Indiana, USA, 2002.