3-2006

Crosscutting score: An indicator metric for aspect orientation

Subhajit DATTA
Singapore Management University, subhajitd@smu.edu.sg

Crosscutting Score- An Indicator Metric for Aspect Orientation

Subhajit Datta
Department of Computer Science and
School of Computational Science
Florida State University
Tallahassee, FL 32306, USA
datta@cs.fsu.edu

ABSTRACT

Aspect Oriented Programming (AOP) provides powerful techniques for modeling and implementing enterprise software systems. To leverage its full potential, AOP needs to be perceived in the context of existing methodologies such as Object Oriented Programming (OOP). This paper addresses an important question for AOP practitioners – how to decide whether a component is best modeled as a class or an aspect ? Towards that end, we present an indicator metric, the *Crosscutting Score* and a method for its calculation and interpretation. We will illustrate our approach through a sample calculation.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, process metrics*

General Terms

Algorithms, Management, Measurement, Design

Keywords

Aspects, Analysis, Design, Metrics

1. INTRODUCTION

Aspect Oriented Programming (AOP) has had several descriptions; from the prosaic – another programming technique – to the poetic – a whole new paradigm of software development. In the fall of 2003, Gregor Kiczales described the then current state of AOP as "moving from the invention phase to the innovation phase"[1]. Two years prior, in an article evocatively titled *Through the looking glass*, Grady Booch had identified AOP as one of the most exciting emergent areas, reflecting, "AOP, in a manner similar to classic patterns and Krutchen's architectural views, recognizes that there are abstractions on a plane different than object-oriented abstractions, which in turn are on a plane different than our executable systems"[2].

AOP stands poised at an interesting juncture today. Its power and grace are proven, best minds of the discipline are delving into it, it is often hailed as the greatest thing to happen to software engineering since object orientation.

It is often easy to appreciate the elan of a new approach in the abstract; still practitioners need and seek guidelines to get them started in the concrete. Under project pressures, the leeway to bridge the cognitive gap in understanding a technology and placing it in context is usually absent. Initial explorations of AOP are often marked by recurrent confusions about when to opt for an AOP solution vis-a-vis an OOP one.

The basic question often asked is: when to use an aspect rather than a class ?

This inspires the search for a metric to quantify the parameters on which such decision can be based. In this paper we propose a *rule of thumb* centering around the *Crosscutting Score* to help best decide what needs to be *aspectualized*, i.e. what is *to be or not to be* an aspect.

In the next sections we discuss the context of AOP in its connections to existing themes and recent research, followed by a reflection on the different *coordinates* of aspect technology. Next, the motivations for a thumb rule are introduced. We then derive the *Crosscutting Score* metric and illustrate its use. The Conclusion summarizes ideas presented in this paper and their relevance to software development.

2. THE CONTEXT OF AOP

AOP gives a novel insight into the eternal issues of analyzing, designing, building and maintaining software systems. Like all successful innovation, AOP seeks solutions to problems that have been known to exist, only brought into recent focus as software engineering grapples with deeper complexity. [1]

Many of the problems for which AOP promises better solutions were and are being *worked around* by existent methods. Designers and developers are often faced with the conundrum – when would the AOP-OOP combination offer

[1]As Grady Booch says so feelingly, "This stuff is fundamentally, wickedly hard – and it's not going to get any better in my lifetime, and I plan on having a long life" [3].

ACM SE'06 March 10-12, 2006, Melbourne, Florida, USA

better returns than conventional OOP; is a functionality best modeled through a class or an aspect ?

This is a fundamental question all users of AOP face; and there are no ready answers.

AOP focuses on situations that have been in limelight ever since programming graduated to software engineering. *Separation of concerns* (and the criteria thereof) has been of primary interest from the time understanding various facets of the problem domain became a nontrivial task. In a paper older than thirty years, (that has aged with amazing grace) Parnas [4] "discusses modularization as a mechanism for improving the *flexibility* and *comprehensibility* of a system while allowing the *shortening of its development time*." He goes on to clarify " 'module' is considered to be a *responsibility assignment*..." (Italics ours.) Every software engineering methodology has arrived with covenants of making systems simpler to understand, easier to extend and faster to construct. Responsibility assignment remains a key factor for achieving these goals, to the extent it has been called a "desert-island skill" [5] – the one critical ability that successful software development must harness.

To be able to decide which component does what, the foremost step is understanding the gamut of activities (*services*, in recent terminology) expected from the system. The word *concern* is often taken to connote the different behaviors of components that collaboratively deliver the system's functionality.

3. RECURRENT MOTIFS AND RELATED WORK

Modularization of crosscutting concerns is often a theme first introduced to AOP beginners [6]. This is indeed a central motif of AOP, and it underscores the links of AOP to some long-circulating ideas in software engineering. At a high level of abstraction, crosscutting concerns can be viewed as behavior such as logging, exception handling, security, instrumentation etc. that stretch across conventional distributions of responsibility. In standard (i.e. non-AOP) OO implementations, such behavior is achieved by specialized classes, whose methods are invoked as required. If at ten different locations in a body of code logging is needed, there will be ten statements where some *log* method of a *Logger* class is called. AOP provides a mechanism to encapsulate such dispersed functionality into modules. Logging et al. are not the only supposedly *peripheral* concerns AOP handles. Aspects can be used to enforce a *Design by Contract* style of programming, a number of OO *design patterns* also have crosscutting structure and can be implemented in a modular and reusable way using aspects [1], [7]. In addition, there is scope for utilizing aspects to deal with the *business rules* – often the most capricious and complex parameters of a system.

Lopes highlights this positioning of aspects vis-a-vis objects as "Aspects are software concerns that affect what happens in the Objects but that are more concise, intelligible and manageable when written as separate chapters of the imaginary book that describes the application" [12].

Several recent studies have explored the feasibility of AOP solutions in different locales and levels of software development. Zhang and Jacobsen present middleware refactoring techniques using aspects [13]. Use of aspects in specific application areas are highlighted in [15], [16]. Design Structure Matrix (DSM) and Net Options Value (NOV) approaches are used in [14] to analyze the modularity of aspect oriented designs.

Although these papers provide valuable insight into the applicability of AOP, we believe a basic confusion continues to assail practitioners, when and why a departure from conventional OOP to AOP will be beneficial. The following sections introduce a mechanism to clarify such concerns.

4. ASPECT ORIENTATION – DIFFERENT COORDINATES

As an evolving technology, we may perceive Aspects in the following lights.

Aspects : Ideation – As an *idea* aspects are precisely what the word "aspect" means, a *way of looking at things* or *how something appears when observed*. In software contexts that translates to looking at the functionality of a system for common behavior that can be isolated. A *method* or a *function* of programming languages is one way of *aspectualizing*, it embodies behavior that is encapsulated and can be invoked by a *method call*; thus localizing the code that implements the behavior.

Aspects : Incarnation – Formalizing ways of discovering, understanding and using aspects as a software development artifact, *incarnates* aspects into AOP. The acronym AOSD (Aspect Oriented Software Development) is somewhat misleading; there seems a hint *aspect orientation* is a whole new methodology of software development, to be preferred over existing techniques. AOP serves to *complement* other models of software development – since OOP is the dominant paradigm of the day, most AOP tool extensions are OO tools [1],[2]. An aspect needs not necessarily be associated with code, *aspectual requirements* [8] or *crosscutting requirements* [9] represent approaches for identifying concerns from the requirement gathering phase. Jacobson presents interesting ideas on how use cases and aspects can "seamlessly" work together [10].

Aspects : Implementation – Aspects are *implemented* through tools and frameworks which provide the *hooks* by which aspect technology is attached to application code, and ensures the combination works as a cohesive unit. *AspectJ* has been the oldest of such tools, which recently joined hand with another implementation, *AspectWerkz* to align their features [11].

The procedure presented in this paper aids the incarnation and implementation of aspects. Figure 1 shows how our *thumb rule* positions amongst these perspectives.

5. A THUMB RULE - IMMEDIATE MOTIVATIONS

One of the earliest lessons one learns from AOP is that it is best to identify aspects early. The *weaving* facilities offered in aspect implementations sometimes give an impression – mostly to starters – that AOP is a mechanism for adding functionality that was not envisaged *a priori*, or to accommodate later needs, such as trace logging or performance monitoring. AOP offers rich set of features for affecting program flow: "Pointcuts and advice let you affect the dynamic execution of a program; introduction allows aspects to modify the static structure of a program" [6]. However, arbitrary use of these abilities has the danger of making software, in Brooksean terms, more *invisible* and *unvisualizable*.

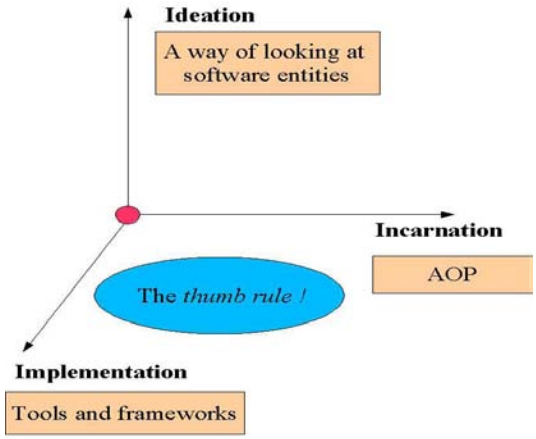

Figure 1: Different views of aspects

Rashid [8] advocates an approach for separating the specification of *aspectual* vs. *non-aspectual* requirements. [9] argues the *problem world* "is often the most appropriate source for early identification of concerns but not necessarily of aspects." [10] suggests strong correlation between use case *extensions* and aspects, seeing an equivalence between *extension points* and *join points*. These outlooks underscore thinking in aspect terms early in the development life cycle.

As noted earlier, responsibility assignment is of central importance in software design. One established canon of OOAD is to have each class fulfill one primary responsibility. Sound design, object oriented or classical, draws on some basic principles. *Cohesion* and *Coupling* – the *yin* and *yang* of software engineering – are some such; each class doing one principal task promotes a design where components function in relative independence, yet generating enough synergy to deliver the user's requirements.

In our discussion, we use *component* to mean an unit of code that is in charge of a chief activity; other ancillary tasks expected from it are deemed secondary. We seek to have a structured way of deciding what best models a component, a class or an aspect, based on the theme of responsibility delegation.

During analysis, techniques such as *noun-analysis*, *CRC cards* help identify components that will be given specific responsibilities. These are yet at a very high level, sometimes referred to as *coarse-grained*, to be refined as development proceeds. But identifying these components is a vital exercise, marking the interface between analysis and design.

6. CROSSCUTTING SCORE

Let $\Theta(n) = (C_1, C_2, C_3, ..., C_m, ..., C_n)$ represent the set of n components for a system. To each component C_m, $(1 \leq m \leq n)$, we attach the following *properties*. A *property* is a set of zero, one or more components.

- *Core* - $\alpha(m)$
- *Non-core* - $\beta(m)$
- *Adjunct* - $\gamma(m)$

$\alpha(m)$ represents the set of component(s) required to fulfill the primary responsibility of the component C_m. As already noted, sound design suggest the component itself should be in charge of its main function. Thus, $\alpha(m) = \{C_m\}$.

$\beta(m)$ represents the set of component(s) required to fulfill the secondary responsibilities of the component C_m. Such tasks may include utilities for accessing a database, date or currency calculations, logging, exception handling etc.

$\gamma(m)$ represents the component(s) that guide any conditional behavior of the component C_m. For a component which calculates interest rates for bank customers with the proviso that rates may vary according to a customer *type*, an *Adjunct* would be the set of components that determine a customer's *type*.

We define,
$\Omega(m) = \beta(m) \cup \gamma(m)$
$cs(m) =$ *Crosscutting Score* of C_m.

Given $\Theta(n)$, $cs(m)$ is computed as follows,

$$cs(m) = \sum_{k=1}^{n} i(m,k)$$

where,

$$i(m,k) = \begin{cases} 1 & \text{if } \alpha(m) \cap (\beta(k) \cup \gamma(k)) \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

The value of $cs(m)$ for a component, relative to those of other components, indicates whether it is a candidate for an aspect.

We now examine the implications of this statement in an example scenario.

7. PUTTING IT TO WORK

We consider an internet banking application. Our *system* allows customers to view their transaction details online, transfer funds between accounts, and print statements in either spreadsheet or text format. Customers are classified into two types, *silver* and *gold* depending on business rules which the bank's management revises from time to time. Some of the application's features are exclusive to *gold* customers; the user interface needs to vary accordingly.

Table 1 shows the components identified with their primary responsibilities.

It may noted the core functionality of a component can be among the non-core ones of another component. This is most apparent in cases such as Logging and Exception handling. But more obscure interaction occurs between User Interface/Transaction Handling with Customer Type Identification; either of the former has conditional behavior based on the functionality of the latter .

Table 2 calculates $\alpha(m)$, $\beta(m)$, $\gamma(m)$, $\Omega(m)$ and $cs(m)$. (Φ denotes a set with zero elements.) As an example, for C_3, $\alpha(3) = C_3$, $\beta(3) = C_5, C_6, C_7, C_{12}, C_{13}$, since *Currency calculation* requires *Logging*, *Instrumentation*, *Exception handling*, *Logging level determination* (it is useful to control the granularity of detail that must be logged; i.e. a mechanism is needed to turn "on" or "off" respective logging levels), *Performance report generation* (while tuning the system at the time of delivery, performance reports based on specific criteria helps discover *bottlenecks* faster). $\gamma(3) = \Phi$, as *Currency Calculation* has no conditional behavior based on customer

Table 2: Calculation of *Crosscutting Score*

C_m	$\alpha(m)$	$\beta(m)$	$\gamma(m)$	$\Omega(m)$	$cs(m)$
C_1	C_1	$C_5, C_6, C_7, C_8, C_9, C_{10}, C_{12}, C_{13}$	C_{10}	$C_5, C_6, C_7, C_8, C_9, C_{10}, C_{12}, C_{13}$	0
C_2	C_2	$C_5, C_6, C_7, C_{10}, C_{11}, C_{12}, C_{13}$	C_{10}	$C_5, C_6, C_7, C_{10}, C_{11}, C_{12}, C_{13}$	0
C_3	C_3	$C_5, C_6, C_7, C_{12}, C_{13}$	Φ	$C_5, C_6, C_7, C_{12}, C_{13}$	0
C_4	C_4	C_7, C_{11}	Φ	$C7, C_{11}$	0
C_5	C_5	C_7	C_{12}	C_7, C_{12}	7
C_6	C_6	C_7	C_{13}	C_7, C_{13}	6
C_7	C_7	C_5, C_{12}	Φ	C_5, C_{12}	12
C_8	C_8	$C_5, C_6, C_7, C_{12}, C_{13}$	Φ	$C_5, C_6, C_7, C_{12}, C_{13}$	1
C_9	C_9	C_7	C_{10}	C_7, C_{10}	1
C_{10}	C_{10}	$C_4, C_5, C_6, C_7, C_{11}, C_{12}$	Φ	$C_4, C_5, C_6, C_7, C_{11}, C_{12}$	3
C_{11}	C_{11}	$C_5, C_6, C_7, C_{12}, C_{13}$	Φ	$C_5, C_6, C_7, C_{12}, C_{13}$	3
C_{12}	C_{12}	C_7	Φ	C_7	8
C_{13}	C_{13}	C_7	Φ	C_7	6

Table 1: Components and the primary responsibilities

Component	Primary Responsibility
C_1	User interface
C_2	Transaction handling
C_3	Currency calculation
C_4	Data storage
C_5	Logging
C_6	Instrumentation
C_7	Exception handling
C_8	User input verification
C_9	Print formatting
C_{10}	Customer type identification
C_{11}	Data access
C_{12}	Logging level determination
C_{13}	Performance report generation

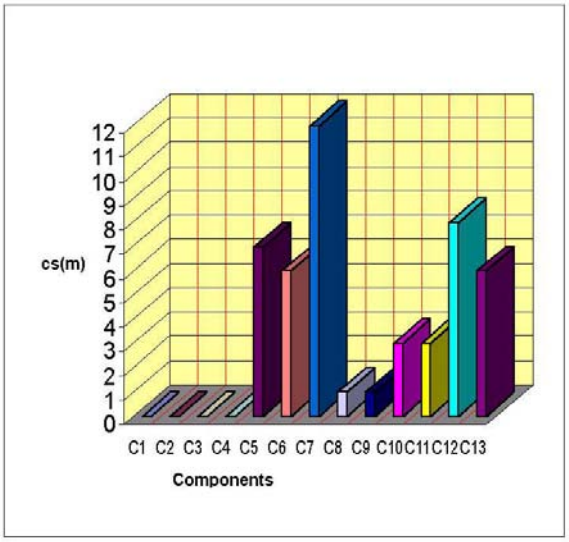

Figure 2: Components vs. Crosscutting Score

type. Hence $\Omega(3) = \beta(m) \cup \gamma(m) = C_5, C_6, C_7, C_{12}, C_{13}$. Applying the algorithm given earlier, $cs(3) = 0$. (Intuitively, currency calculation has a *localized* concern, having no interaction with other components.)

Similarly, for C_6, $\alpha(6) = C_3$, $\beta(6) = C_7$, $\gamma(6) = C_{13}$, since instrumentation is guided by the criteria of performance report generation, conditionally measuring some parameters over others. $\Omega(6) = C_7$, C_{13} and $cs(6) = 6$. *Crosscutting Score* of 6 for the component indicates its core functionality is being used across some other components – intuitively, instrumentation is needed for all components with nontrivial processing. This value of $cs(m)$ makes it a suitable to be modeled as an aspect rather than a class.

We plot C_m vs. $cs(m)$ in Figure 2. This graph serves as the basis for deciding whether a component may be a class or an aspect. The components with higher $cs(m)$ values have primary behavior that is *crosscutting* – AOP offers great benefits if they are aspectualized. The ones with lower values deliver relatively isolated functionality, classes suffice their implementation.

But a key question remains, what is the threshold between *high* and *low* values of $cs(m)$?

8. INTERPRETING THE RESULTS

Software design is subjective – we draw upon experience and intuition to reach decisions. Calculation and survey of the $cs(m)$ values streamline the process to a certain extent, helping designers select one option over another. In our case, C_7, C_{12}, C_5 , C_6 and C_{13} are clearly *aspectual*. Between C_{10} and C_{11}, the former encapsulates business logic while the latter, data access mechanisms. Business logic is usually prone to change and future enhancements may need wider application of business rules; activities for accessing databases (opening/closing connections, connection pooling etc.) are relatively less volatile. It is reasonable to model C_{10} as an aspect and C_{11} as a class, even though they have the same *Crosscutting Score* value. C_1, C_2, C_3, C_4, C_8, C_9, definitely on the lower side of $cs(m)$ range, are clearly classes.

Thus, there is no "cutoff" $cs(m)$ value to segregate components into classes and aspects. While some components will

be clear aspirants one way or the other, for the *borderline* ones, the designer's judgment comes into play.

A few subtleties are worth pointing out. Components implementing logging, exception handling, database access are easy to pick as potential aspects – their functionality *stretches across* the application – the $cs(m)$ values calculated above also support such observations. However it is less obvious *Customer type identification* may also be *aspectualized*. The component decides whether a customer is *silver* or *gold* (or even some other metal of commensurate nobility, should there be more categories later). Calculating the $cs(m)$ helps in discovering such *covert* aspects.

As emphasized earlier, our algorithm is a judgment aid for designers. The *ranking* of the components based on respective $cs(m)$ is of lesser importance than recognizing the relative distribution of the *Crosscutting Scores*. The $cs(m)$ is one pointer in reaching an overall expedient design involving classes and aspects.

The *thumb rule* is summarized as,

- Identify components based on their primary (core) functionality.
- Calculate *Crosscutting Score* $cs(m)$ for each component.
- Relatively higher $cs(m)$ value signifies crosscutting functionality – the corresponding component is a strong aspirant for an aspect.
- Based on $cs(m)$ value and other design desiderata, model each component as an aspect or a class.

The choice of the phrase *thumb rule* has been deliberate; this is a heauristic rather than a formula. The software engineering community continues its quest for sure-shot recipes of design *nirvana*.

9. CONCLUSION

AOP is not a revolutionary doctrine. It is one more step in the evolutionary quest for simple and elegant foundations to build complex software. Effective use of AOP happens when it is successfully integrated, *gelled* as it is sometimes colorfully called, into extant tools and techniques.

This paper introduces an approach for deciding whether a piece of functionality is best abstracted in an aspect or a class. The *thumb rule* centering around the *Crosscutting Score* will assist the design of solutions best suited to AOP's reach and context.

10. ACKNOWLEDGMENTS

I would like to thank Dr. R. van Engelen for assistance in preparing this paper. This work is supported in part by the Department of Energy grant DEFG02-02ER25543.

11. REFERENCES

[1] G. Kiczales. Interview with Gregor Kiczales. Topic : Aspect Oriented Programming (AOP). *www.theserverside.com*, July 2003.

[2] G. Booch. Through the Looking Glass. *www.sdmagazine.com*, July 2001

[3] G. Booch. The Complexity of Programming Models. *AOSD '05*, Chicago, USA, March 2005.

[4] D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, December 1972, Volume 15, Number 12.

[5] C. Larman. *Applying UML and Patterns : An Introduction to OOA/D and Iterative Development, 3rd Edition*. Prentice Hall, 2005.

[6] N. Lesiecki. Improve modularity with aspect-oriented programming. *IBM developerWorks*, January 2002.

[7] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. *AOSD '05*, Chicago, USA, March 2005.

[8] A. Rashid, A. Moreira, J. Araujo. Modularization and Composition of Aspectual Requirements. *AOSD '03*, Boston, USA, 2003.

[9] B. Nuseibeh. Crosscutting Requirements. *AOSD'04*, Lancaster, UK, March 2004.

[10] I. Jacobson. Use Cases and Aspects Working Seamlessly Together. *Journal of Object Technology*, vol. 2, no. 4, July-August 2003.

[11] D. Sosnoski. Classworking toolkit: Putting aspects to werk. *IBM developerWorks*, March 8, 2005.

[12] C.V Lopes. Aspect-Oriented Programming : An Historical Perspective. *ISR Technical Report UCI-ISR-02-5*, www.isr.uci.edu/tech-reprt.html, December, 2002.

[13] C. Zhang, H. Jacobsen. Refactoring Middleware with Aspects. *IEEE Transactions On Parallel and Distributed Systems*, Vol. 14, No. 11, November 2003.

[14] C.V Lopes, S.K. Bajracharya. An Analysis of Modularity In Aspect Oriented Design, *AOSD '05*, Chicago, USA, March 2005.

[15] E. Putrycz, G. Bernard. Using Aspect Oriented Programming to Build a Portable Load Balancing Service. *Proc. 16th Int'l Conf. Distributed Computing Systems Workshops*, 2002.

[16] M. Kersten, C. Murphy. Atlas: A Case Study in Building a Web-based Learning Environment Using Aspect-Oriented Programming. *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and applications*, 1999.