An Introduction to Aspect-Oriented Software Development

Pouria Shaker Dennis K. Peters

Electrical and Computer Engineering
Faculty of Engineering and Applied Science
Memorial University of Newfoundland

pouria@engr.mun.ca **dpeters@engr.mun.ca**

October 14, 2005

Abstract

Separation of concerns (SOC) has been identified as one of the core principles of software engineering. It is desirable to decompose a system into modules in such a way that modules responsible for a common concern are tightly coupled and modules responsible for different concerns are loosely coupled. Over the years software development technology has evolved to achieve a higher level of SOC. Aspect-oriented software development (AOSD) is a post object-oriented technology that helps achieve better SOC by providing mechanisms to localize cross-cutting concerns (e.g. security, synchronization, logging, etc.) in software artefacts throughout the software development process. This paper serves as an introduction to AOSD.

1 Introduction

Over the years, computer hardware and software have evolved hand in hand. In the early days, due to hardware limitations, the problems solved by computers were simple and so was the software written to solve them. Demands for using computers to solve more complex problems led to advancements in hardware technology; software technology grew as a result to support the complex software solutions required for such problems.

Traditional engineering disciplines manage the complexity of systems by identifying and separating the system's *concerns* and treating each concern in isolation; such an approach known as *separation of concerns* (SOC), leads to systems that are easier to implement, verify, evolve, and understand. The craft of software development quickly adopted this approach; programming languages were forerunners in the advancement of software technology with languages progressively providing abstractions such as functions, procedures, abstract data types, and objects to help achieve higher levels of SOC. *Aspect-oriented programming* (AOP) technology is a possible next step in this progressive trend.

Modern software development or software engineering is more than just coding; it is an iterative process made up of several stages with methodologies to guide each stage and tools to support the methodologies. Programming languages are merely tools in the development process, but traditionally most advancements have first been made in this department and later extended to the entire development process. Aspect-oriented technology is no exception; following the development of AOP languages, aspect-oriented ideas extended to the entire development process forming the field of *aspect-oriented software development* (AOSD).

The remainder of the paper is organized as follows: section 2 describes the limitations of the currently dominant programming technology in achieving SOC; section 3 presents AOP as a possible solution to these limitations;

section 4 presents AOSD as an extension of aspect-oriented ideas to the entire development cycle; section 5 presents the aspect-interaction problem as a fertile area for further research; the final section presents conclusions to this discussion. This paper assumes basic knowledge in software engineering and object-oriented technology.

2 Object-oriented programming and SOC

In 1972, Parnas [1] argued that to create systems that are easy to implement, understand, verify, and evolve, one must decompose the system into modules in such a way, that each module hides an aspect of the system that can evolve independently of other aspects; this leads to modules that are loosely coupled and can be implemented, understood, verified, and evolved independently. In other words, the independent concerns of the system are to be identified and localized into modules; this is essentially a recipe for achieving SOC.

One consequence of this research was the development of the *object-oriented programming* (OOP) technology. In OOP the problem domain is modeled as a collection of things or *objects*; each object belongs to a class that specifies the behaviour and attributes of its instances; objects collaborate to satisfy the concerns of the system. OOP is currently the dominant programming technology and for good reason: it helps achieve better SOC than its predecessors; but as the following example from [2] illustrates, there is still room for improvement.

A typical OO model for a simplistic figure editing program is depicted in Figure 1. The concerns of representing the display screen and the figures, points, and lines on the screen are localized by the concrete classes `Display`, `Figure`, `Point`, and `Line` respectively. Now consider the concern of updating the display screen each time points or lines move; this concern cannot be localized in a single module in this model; its implementation *cross-cuts* the `Point` and `Line` modules as invocations of `Display.upate()` in each of modifier methods of `Point` and `Line`. In this model, display updating is a *cross-cutting concern*.

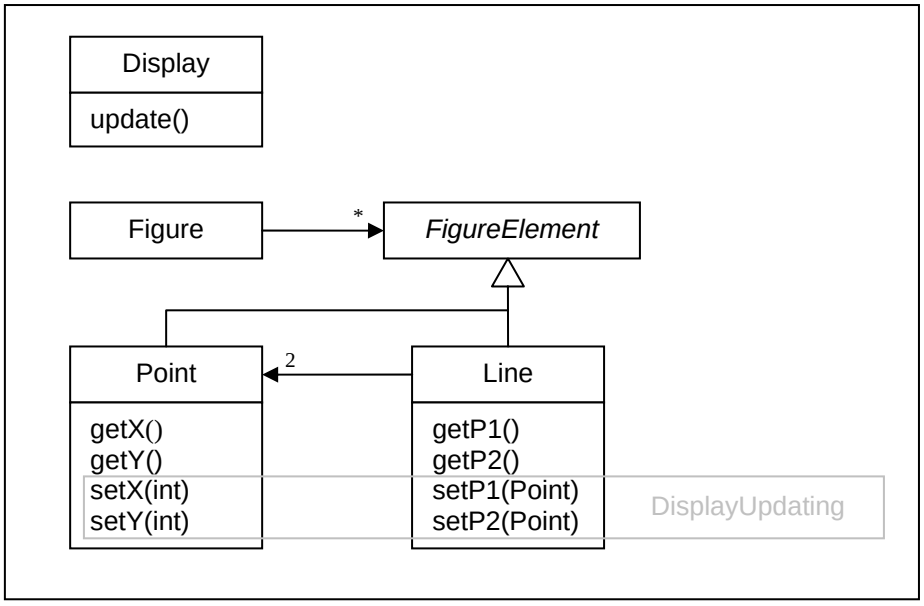

Figure 1: Cross-cutting in OO models

What if we try a different modularization that localizes the display updating concern? We will sadly discover that this will leave other concerns scattered across the new model. The cross-cutting nature of concerns is an inherent property of many real problems and OO technology falls short in localizing all concerns in such problems; let us see why.

As shown in Figure 2 adopted from [3], the concern space of many problems is multi-dimensional. In OOP, the system is modularized across a single dimension; all concerns along this dimension are neatly localized in the OO

model while the remaining concerns cross-cut the model. This is the result of mapping a multi-dimensional concern space onto a single-dimensional implementation space.

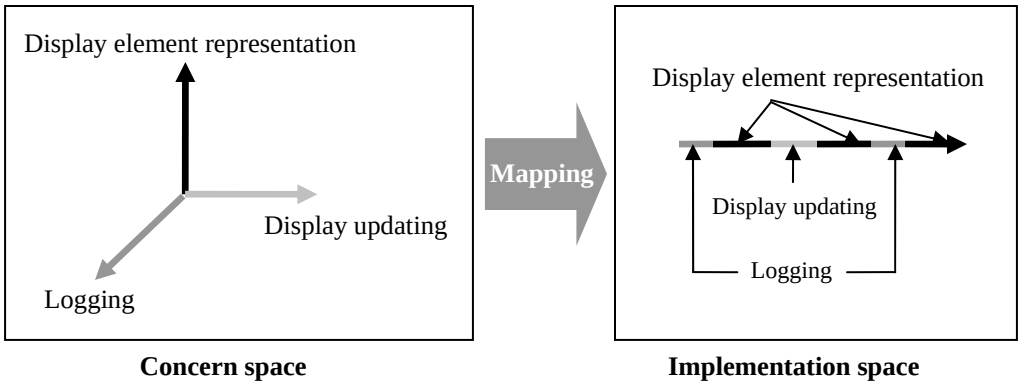

Figure 2: Mapping of a multi-dimensional concern space onto a single-dimensional implementation space

In our example, the concerns of representing the display screen and figures, points, and lines fall on the "display element representation" dimension of Figure 2; the display updating and logging (added to emphasize the multi-dimensional nature of the concern space) concerns fall on the remaining orthogonal dimensions.

The inability of OO technology to simultaneously localize orthogonal concerns has its consequences: cross-cutting concerns are implemented in several modules (*scattering*) and a single module implements more than one concern (*tangling*); these are signs of poor modularity: scattering leads to poor traceability from cross-cutting concerns to their implementation; tangling hinders ease of module implementation (one has to focus on multiple concerns while implementing a module), comprehension, and reuse (the implementation of one concern comes with the baggage of other concerns); it also becomes hard to evolve the system since implementing an additional cross-cutting concern involves modifying multiple modules.

3 Aspect-oriented programming

Several post-object programming (POP) technologies emerged to address the limitation of OO technology in achieving SOC across more than one dimension; these include adaptive methods (Lieberherr [4]), subject-oriented programming (Ossher and Tarr [5]), composition filters (Bergmans and Aksit [6]), and aspect-oriented programming (Xerox Palo Alto Research Center [7]). These related research paths have now converged under the title of *aspect-oriented programming* (AOP).

Despite ongoing and productive dialogue amongst the AOP community, a common consensus on what constitutes an AOP approach is yet to be reached (though significant efforts have been made [8, 9, 10]). Perhaps the most widely cited endeavour to characterize AOP is that of Filman [10]: that AOP is *quantification* and *obliviousness*. Quantification means that programs can include *quantified statements* (i.e. statements that apply to more than one place) of the form "In programs P, whenever condition C arises, perform action A"; obliviousness means that authors of a program P need not be aware of quantified statements that reference them.

How do quantified statements help? Figure 3 illustrates how the display updating concern from the figure editing example of section 2 can be localized in a quantified statement (another quantified statement could localize the logging concern); notice how the authors of the `Point` and `Line` classes can be oblivious of the display updating concern (or other cross-cutting concerns such as logging) and focus on implementing the concerns of representing points and lines.

In general given an N-dimensional concern space and an M-dimensional implementation space where M < N, cross-cutting concerns can be localized in quantified statements in an AOP system (this supports the notion that AOP does

not replace existing technologies, rather it complements them); this improves modularity with the following implications:

- *Improved traceability*: cross-cutting concerns can be easily traced to quantified statements.
- *Ease of implementation and comprehension*: authors/readers of modules can focus on implementing/understanding one concern and can be oblivious of cross-cutting concerns.
- *Module reusability*: modules implement a single concern and do not come with the baggage of other concern implementations.
- *Improved evolvability*: adding a cross-cutting concern is simply a matter of adding a quantified statement.

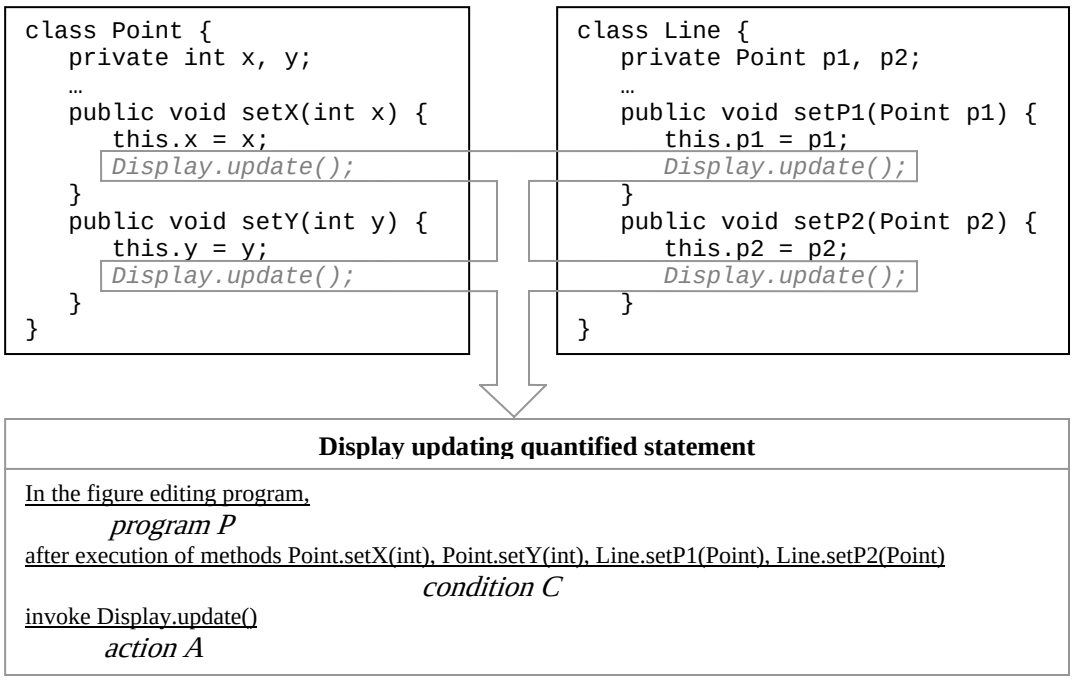

Figure 3: Use of quantified statements in the figure editing example

According to [10], to implement an AOP language (i.e., a language that allows quantified statements over oblivious programs) one must consider three issues:

1. *Quantification*: what conditions can we use in quantified statements? In other words, to what points in the execution of programs can actions be tied? Two broad types are:
    a. Points that can be specified by elements of the static structure of programs (e.g. method calls which can be specified by method signatures)
    b. Points that depend on run-time behaviour (e.g. size of the call stack)
2. *Interface*: how do quantified statements communicate with programs and with other quantified statements?
3. *Weaving*: what mechanism interleaves the execution of actions in quantified statements with the execution of affected programs?

AspectJ [11] is a general purpose aspect-oriented extension to Java developed by the AOP group at Xerox Palo Alto Research Center (PARC) and is perhaps the most popular existing AOP language. AspectJ allows writing quantified statements over conventional Java programs; quantified statements are specified by class-like constructs called *aspects* (note that the term aspect-oriented programming was coined by Gregor Kiczales of Xerox PARC). Like classes, aspects can have attributes and methods; in addition they encapsulate the remaining components of quantified statements: conditions are specified by *pointcut expressions* and actions are specified by method like constructs called *advice*.

Let us see how AspectJ addresses the three implementation issues listed above:

1. *Quantification*: AspectJ has a *join-point model* that determines the types of *join-points* (i.e., points in the execution flow of programs) that aspects can advise. These include method or constructor calls and executions, advice executions, static class initializations, object or aspect initializations, field read or write accesses, and exception handler executions. Sets of join-points are specified by pointcut expressions and each piece of advice has an associated point-cut expression; advice can be specified to execute before, after, or around join-points.
2. *Interface*: aspects can gain contextual information of the join-points they advise; this is done using parameterized point-cut expressions. Additionally aspects can introduce fields and methods into other types through the *inter-type declaration* mechanism.
3. *Weaving*: the AspectJ compiler (ajc), combines Java and aspect source files and jar files into woven class files or jar files.

Figure 4 shows an AspectJ aspect written for the display updating concern of our running example.

```
aspect DisplayUpdating {
   pointcut move():
      execution(public void Point.setX(int)) ||
      execution(public void Point.setY(int)) ||
      execution(public void Point.setP1(Point))||
      execution(public void Point.setP2(Point));

   after(): move() {
      Display.update
   }
}
```

Pointcut expression

advice

Figure 4: Display updating aspect

Other AOP languages such as the DJ library [4], Hyper/J [5], and composition filters [6] use different approaches to address the three implementation issues; the interested reader is referred to the cited sources for details.

4 Aspect-oriented software development

Software development has evolved from a programming activity to a full-blown engineering process; modern software engineering constructs systems using processes that progressively refine higher-level abstractions of the system to lower-level abstractions starting from requirements and stopping at executable code. Preserving two important properties across this refinement process helps a great deal in producing high-quality software: *modularity* and *traceability*. The benefits of modularity were discussed in previous sections. Both modularity and traceability are crucial in managing change in systems; when the system changes at a given level of abstraction, modularity ensures that the change is localized, and traceability ensures that the change can be propagated naturally and easily to other levels of abstraction.

In previous sections AOP was described as a technique that improves modularity at the code level; the benefit of applying the AO methodology to earlier stages of the software development cycle is two-fold: first it ensures improved modularity at all stages of the development process; secondly preserving the notion of aspects throughout the development process ensures traceability. These ideas launched the field of aspect-oriented software development (AOSD) with an active research community [12]. As stated in [13], the same way that AOP extends conventional programming technology, AOSD extends conventional software development practices.

An excellent survey of research aimed at applying AO techniques to various stages of the development process including requirements engineering, specification, design, implementation, and evolution is given in [14] ; [14] also provides guidelines for choosing from existing approaches depending on the developer's need.

5 The aspect interaction problem

In an AO implementation of a system, the system's concerns (cross-cutting or otherwise) are neatly localized into modules; each concern represents a goal in the system that we wish to reach. An important question to answer is whether these goals *interact*; i.e., does the satisfaction of one goal hinder the satisfaction of others? In AOP literature a narrower version of this problem is known as the *aspect-interaction* problem: here the question is whether the introduction of an aspect (a module that localizes a cross-cutting concern) creates interactions with other modules of the system; this is reminiscent of the *feature interaction* problem in the telecommunications industry where we wish to know whether adding a feature (such as call-waiting, call-forwarding, etc.) to the base phone service creates interactions with other features or with the base phone service itself [15]. We can present the aspect interaction problem as follows (here we refer to modules that localize cross-cutting concerns as aspects and the collection of all other modules as the *base system*):

- Let P1 be a property satisfied by aspect A.
- Let P2 be a property satisfied by the composition of the base system and n aspects (SYS).
- Both P1and P2 should remain satisfied by the composition of A and SYS; if not then there is an interaction.

In [14] the aspect-interaction problem has been identified as an under-researched area in AOSD; a few efforts to attack this problem are listed in [14] the most notable of which are [16], which presents a formal AOP language with corresponding interaction detection and resolution analyses and [17], which proposes to use program slicing to extract manageable models from AspectJ code and to use these models to prove properties of the system including absence of interaction. We wish to investigate practical approaches to support the formal detection of aspect interactions at the design phase; the main components of this research are the identification of suitable techniques for modeling aspects at the design phase and tools to perform formal analyses on these models.

6 Conclusion

Aspect-oriented software development (AOSD) is an emerging technology that supports multi-dimensional separation of concerns throughout the software development cycle. The aspect-interaction problem has been identified as an under-researched area in AOSD; we wish to investigate practical and formal approaches for the detection of aspect interactions at the design phase.

References

[1] D. L. Parnas, 1972. On the criteria to be used in decomposing systems into modules. Comm. ACM 15, 12 (Dec.), 10531058.
[2] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of AOP. Communications of the ACM, 44(10):33--38, Oct. 2001.
[3] R. Laddad, 2003. AspectJ in Action. Manning Publications, Greenwich, Connecticut.
[4] K. Lieberherr, D. Oleans, and J. Ovlinger. Aspect-Oriented programming with adaptive methods. Communications of the ACM, 44(10):39--41, Oct. 2001.
[5] H. Ossher, and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. Communications of the ACM, 44(10):43--50, Oct. 2001.
[6] L. Bergmans, and M. Aksit. Composing crosscutting concerns using composition filters. Communications of the ACM, 44(10):51--57, Oct. 2001.
[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, June 1997. Springer-Verlag.
[8] H. Masuhara and G. Kiczales. Modeling crosscutting in aspectoriented mechanisms. In L. Cardelli, editor, Proceedings of European Conference on Object-Oriented Programming (ECOOP2003), volume 2743 of Lecture Notes in Computer Science, pages 2--28, Darmstadt, Germany, July 2003. Springer-Verlag.
[9] M. Monga "On aspect-oriented approaches", Proceedings of the european interactive workshop on aspects in software (EIWAS'04) Berlin, Germany, 2004

[10] R. E. Filman, D. P.Friedman: Aspect-Oriented Software Development, chapter 2- Aspect-Oriented Programming is Quantification and Obliviousness, Addison-Wesley, 2004
[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, 2001. Getting started with AspectJ. Comm. ACM 44, 10 (Oct.), 5965.
[12] AOSD, "Aspect-Oriented Software Development", http://aosd.net, 2005.
[13] R. Filman, T. Elrad, S. Clarke, and M. Aksit eds., Aspect-Oriented Software Development, Part 2-Software Engineering, Addison-Wesley, 2004.
[14] G. Blair, L. Blair, A. Rashid, A. Moreira, J. Araújo, R. Chitchyan: Aspect-Oriented Software Development, chapter 17-Engineering Aspect-Oriented Systems, pages 379-406. Addison-Wesley, 2004
[15] F. Beltagui (2003) Features and Aspects: Exploring feature-oriented and aspect-oriented programming interactions. Technical Report No: comp-003-2003, Lancaster University.
[16] R. Douence, P. Fradet, and M. Südholt 2002. A framework for the detection and resolution of aspect interactions. In 1st ACM Conf. Generative Programming and Component Engineering (GPCE), (Pittsburgh), D. S. Batory, C. Consel, and W. Taha, Eds. LNCS, vol. 2487. Springer-Verlag, Berlin, 173188.
[17] L. Blair and M. Monga 2003. Reasoning on AspectJ programmes. In 3rd German Informatics Society Workshop on Aspect-Oriented Software Development (AOSD-GI), (Essen, Germany).