ASPECT ORIENTED PROGRAMMING: A TOOL TO HANDLE CROSS-CUTTING CONCERNS IN MODULARIZATION MECHANISM

Dr. Mahua Banerjee
Department of Information Technology, Xavier Institute of Social Service, Ranchi,

Abstract: Traditional programming languages limit the level of modularity which is essential for creating evolvable software. Some requirements, features, properties cannot be modularized. The code related to these concerns is then scattered and tangled. Such concerns are called cross-cutting concerns which results in difficulties for modularization. AOP aims at modularizing crosscutting concerns into a new kind of module, called aspect. In AOP, non cross-cutting concerns are programmed in a base programming language. Cross-cutting concerns are expressed as aspects. The aspects are then woven into the base program with an aspect weaver which restricts scattering and tangling.

Keywords: Aspect-oriented programming, join-point, pointcut, advice, Inter type declarations

1. INTRODUCTION

This paper focuses on analyzing the role of cross-cutting concerns in software development. Aspect Oriented Programming (AOP) is a novel paradigm which aims at separating and modularizing cross-cutting concerns by means of a new kind of module known as aspects.

Many of the current approaches to software engineering view the development process as being an evolutionary cycle that continues throughout the lifetime of a piece of software. This is in contrast to the classical waterfall approach where the stages of development are partitioned into a strict chronological order that begins with requirements gathering and ends with deployment and maintenance. Under the evolutionary approach software artifacts undergo continual revision and expansion as new requirements are discovered, bugs are found and the external environment changes.

One of the primary goals of software engineering is to assist developers in dealing with this dynamic environment, where a large part of their work is to change existing code rather than to write new code. One of the ways in which this can be done is through better separation of concerns. Programming languages provide mechanisms for dividing programs into modules that represent particular design decisions, features or pieces of functionality which can be generally referred to as concerns. Modularizing concerns helps during evolution because developers do not have to deal with the program in its entirety every time they want to make a change. They can focus on just the modules that relate to their task and the compiler can provide some assurances that implementation details will not have far reaching effects. Modules also provide a means for encapsulating generic functionality so that code can be reused across multiple projects [1].

However, not all concerns can be easily modularized. When a designer chooses a decomposition of a program into modules, he or she does so with the intent of making the expected evolution tasks easier for developers to perform. In practice, finding a decomposition that supports all required evolution tasks is often impossible. In some cases this is due to new requirements or environmental changes that could not have been predicted and so were not planned for. In other cases the programming language used to implement the software does not provide adequate means to modularly express the tangled web of interactions that exist between all the relevant concerns. This leads to the existence of concerns whose implementations are scattered across multiple modules. This mismatch between the chosen decomposition and the required programming tasks is often referred to as the tyranny of the dominant decomposition and is one of the main motivations for aspect-oriented programming (AOP) [2].

Aspect-oriented programming builds on top of object-oriented programming by introducing new forms of modularity. AOP languages provide more flexibility in choosing decomposition through the use of aspects which define both state and behaviour that can be woven into the object-oriented structure of a program [3]. While this improves the coverage of decompositions over evolution tasks, it does not provide a complete solution because there is still only a single decomposition available. AOP approaches make it easier to modularize concerns that were previously scattered amongst object-oriented classes. At the same time they tend to scatter the implementation of classes across aspects. This makes some tasks easier to perform at the expense of making others more difficult.

2. Literature Review

A number of studies will be reviewed in this section to offer a literature overview about the concepts of AOP and its relevance in modular programming.

2.1 CONCERNS AND ASPECTS

Object-Oriented Programming (OOP) is probably the most commonly used programming paradigm today. The evolution from assembler language to current software engineering paradigms reflects the will for better readability and re-usability—or, more generally, better organization— in designing applications. Functional, procedural and object-oriented programming languages have a common way of abstracting and separating out concerns: they rely on explicitly calling subprograms (subroutines, procedure, methods, *etc.*) that represent *functional units* of the system. However, not all concerns can be encapsulated properly in a functional decomposition. For example, tracing and logging are concerns that are usually distinct from the functional units they are related to (i.e. the units whose behavior is traced or logged). As a result, they

must be coordinated with other functional units and they usually involve code scattered throughout several of these functional units. Aspect-Oriented Programming aims at better separation of concerns by providing the *aspect* as a means to encapsulate such crosscutting concerns [4].

2.2. A diagrammatic representation of Aspects crosscutting classes

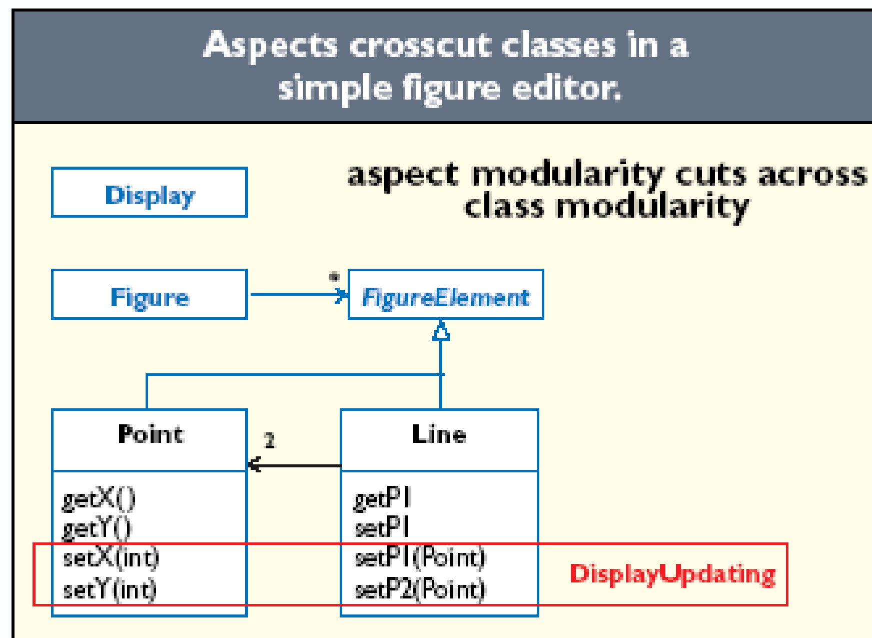

Fig. 1: Example of aspects in a figure editor

A short example of crosscutting concerns and use of AOP is shown in Figure1. The figure shows the classes used in a simple figure editor: "*a* Figure *consists of a number of* FigureElements*, which can be either* Points *or* Lines*. [...]*
There is a single Display *on which figure elements are drawn*". Methods setX and setY of the FigureElements involve two separate actions: they must update the coordinates in their target object and they must trigger the redrawing of the display. Updating coordinates X or Y of the target is intrinsic to the object and clearly corresponds to, respectively, methods setX and setY. However, the concern of updating the display has to be handled after executions of either setX or setY. Thus, the display-update concern crosscuts two methods within the same class. This concern must also be applied in several classes. Updating the display is a concern that is not directly related to the FigureElement model.

One can imagine another system, using the same FigureElement model, which would render elements of this model on a printer instead of a screen. It is possible for all the statements related to the display to be embedded in the code for the FigureElement model. However, in that case, transforming the application so that it uses a printer would require modifications in diverse parts of the code, because the concern of updating the display would crosscut the model and would not be encapsulated in its own entity. An aspect-oriented implementation of this application would encapsulate these kinds of concerns into aspects, making it easier to read, re-use and adapt the code in other contexts. The principles of such an implementation are described in the next section.

2.3 PROGRAM STATEMENTS

Aspect-Oriented Programming aims to be able to link together concerns that cut across each other, and yet encapsulate them transparently as separate program entities. The general style of programming that arises out of this aim consists of program statements of the form:
"*In programs* P*, whenever condition* C *arises, perform action* A*.*"

Based on the above problem statement we propose that AOP is an extension of OOP. In most AOP languages the concept of an aspect extends the concept of a class. Besides structural elements known from OOP, e.g., methods and fields, aspects may contain also join points, pointcuts, advice, and inter-type declarations [5, 6, 7].

2.4. Join point

A *join point* is a point in the structure or in the execution of a program where a concern crosscutting that part of the program might intervene. Join points are the points that can be used to express potential conditions C in programs, according to the above formulation. Join points can be seen as hooks in a program where other program parts can be conditionally attached and executed. The primary mechanism of AOP is the extension of events occurring at runtime, so-called join points. The static representation of a runtime event is called join point shadow. Join point shadows are for example statements of method calls, object creation, or member access.

2.5 Pointcut

A *pointcut* is a subset of all possible join points. The expression of a pointcut is the *pointcut descriptor* (often, the term "pointcut" is used in place of "pointcut descriptor"). A pointcut descriptor defines the condition C in the above formulation. This condition matches a subset of join points which is the pointcut. In other way a pointcut is a declarative specification of the join points that an aspect will be woven into, i.e., it is an expression (quantification) that determines whether a given join point matches.

2.6 Advice

The piece of code A that is to be executed when condition C arises (i.e. at a join point of the pointcut) is called the *advice*. An advice is a method-like element of an aspect that encapsulates the instructions that are supposed to be executed at a set of join points. Pieces of advice are bound to pointcuts that define the set of join points being advised.

3. Model of an aspect and an aspect with a class in programs

3.1 An example of an aspect

```
aspect ExceptionPrinter {
pointcut allmethods() : executions(*  *(..));
  static after() throwing (Exception e)  :  allMethods ( )   {
     System.out.println("Uncaught exception: " + e);
   }
}
```

The above code is a simple example of an aspect that prints all exceptions not handled by any given method in the current package. We could use this code during testing to see whether unexpected errors propagate out of a package, thereby indicating a bug in the package. We might wish this code to be in the final version, logging the error to disc instead of printing it. Users can also have such logs periodically mailed back to the software maintainer (so that bugs in software that went uncovered at testing time would get reported), maintainers would not need to rely on end users to notice and report problems.

In this example, we specify a pointcut named allMethods, which cuts all executions of any function in the current package. The executions keyword specifies that we wish to cut method executions, and the argument is a wildcard expression declaring that we wish to cut all methods with any signature. The first asterisk indicates that matched methods might return anything; the second indicates that any method name should match. The double dot indicates that matched methods might have any number of arguments.

Our pointcut has some advice defined for it when an exception is thrown. Our advice could be specified for multiple pointcuts and hence the pointcut's name is specified after a colon (this name could be a list of names instead).

3.2 Example of Join point, Pointcut and Advice in a program segment

Another example for pointcut and advice along with a class is given below. The aspect MyAspect (Lines 11–22) extends the classes Label and Button. The pointcut LabelChangeCall (Line 12) refers to all statements calling methods of the class Label (call pointcut), e.g., call statements for the method setText. Hence, the according piece of advice of Line 14 (before advice) is introduced into the method click of the class Button before the method Label.setText is invoked (Line 7). Advice can also be applied after (after advice) or around (around advice) a join point. The advice of the pointcut LabelChangeExec (Line 15) refers to the body (execution advice) of the method setText, i.e., the advice is introduced into the method setText of class Label (Line 2).

While pieces of call advice, e.g., LabelChangeCall, are invoked only for referred calls of the method, execution advice, e.g., LabelChangeExec, is applied every time the method is called.

```
1 public class Label {
2 public void setText (){/* . . .*/}
3 }
4 public class Button {
5 public void click (){
6 /* . . .*/
7 myLabel . setText (" Button clicked ")
8 /*  . . . */
9    }
10 }
11 public aspect MyAspect {
12 protected pointcut LabelChangeCall (): cal l (*
Label .*(..) );
13 protected pointcut LabelChangeExec (): execution(*
Label .*(..) );
14 before(): LabelChangeCall (){/* . . . */}
15 before(): LabelChangeExec (){/* . . . */}
16
17 public String Creator . Name ;
18 public void Creator . printName (){/*. . . */}
19
20 public HashMap printer ;
21 public void getPrinter (){/* . . . */}
22 }
```

As per the above program segments we can identify that the unit of code that defines the pointcuts and the advice related to the same concern is called the *aspect*. An aspect can also be more generally defined as a unit that encapsulates a crosscutting concern. We also identify few related terminologies like Inter type declarations and Aspect fields and methods.

Inter type declarations (ITD) are methods or fields that are inserted into OOP classes by an aspect and thus become members of these classes. Additionally, interfaces can be extended with methods and fields. In the above example the aspect MyAspect defines two ITD to insert the field Name (Line 17) and the new method printName (Line 18) into a class Creator. Inter-type declarations are also known as introductions as it injects new members into classes.

Aspects can contain members similar to members of a class, i.e., aspects can contain methods, fields, or inner classes and interfaces [8]. These aspect members can be invoked from code inside the aspect, e.g., by advice, but also from external classes. The aspect MyAspect includes one aspect field and one aspect method (Lines 20–21). If aspect fields and methods are invoked from advice, ITD, or from the classes (using the aspect method aspectOf), and no extra declarations are declared (e.g., percflow), then every reference to aspect members refers to the same single aspect instance, thus the aspect is instantiated once.

Aspects can declare a class to implement additional interfaces. Furthermore, aspects can declare a class to inherit from additional classes.

Other AOP

If a user defined constraint is violated by the classes, the aspect weaver can be instructed to invoke compiler warnings or compiler errors. Precedence declarations define the ordering of advice if join point shadows are advised by more than one aspect.

The counterparts of aspects are *components* or *base code*, which are the functional units of code that do not contain aspect-oriented statements but only base actions.

Components are units of code as written using functional, procedural or object oriented languages. To some extent, the advice in aspects could be considered as a component, within which aspects could intervene. However, the problems that can arise out of interaction between aspects lie beyond this introduction; they are the subject of continuing research.

In the example shown in Figure 1, the Figure Element model is an example of a component. The update of the display is implemented in an aspect. A pointcut descriptor in this aspect expresses the points in the execution flow after returning from methods setX or setY, and the piece of advice associated with this pointcut then updates the display.

Mixing components and aspects together, so that the behaviour specified by the aspects occurs where and when it is supposed to, is the process of *weaving*. Some implementations use a specific type of compiler, called a *weaver*, to generate an executable from components and aspects. Other implementations perform the weaving at runtime or load-time, using mechanisms equivalent to a runtime form of compilation.

With AOP, we start by implementing our project using our OO language (for example, Java), and then we deal separately with crosscutting concerns in our code by implementing aspects. Finally, both the code and aspects are combined into a final executable form using an aspect weaver. As a result, a single aspect can contribute to the implementation of a number of methods, modules, or objects, increasing both reusability and maintainability of the code. Figure 2 explains the weaving process. ? It has to be noted that the original code doesn't need to know about any functionality the aspect has added; it needs only to be recompiled without the aspect to regain the original functionality.

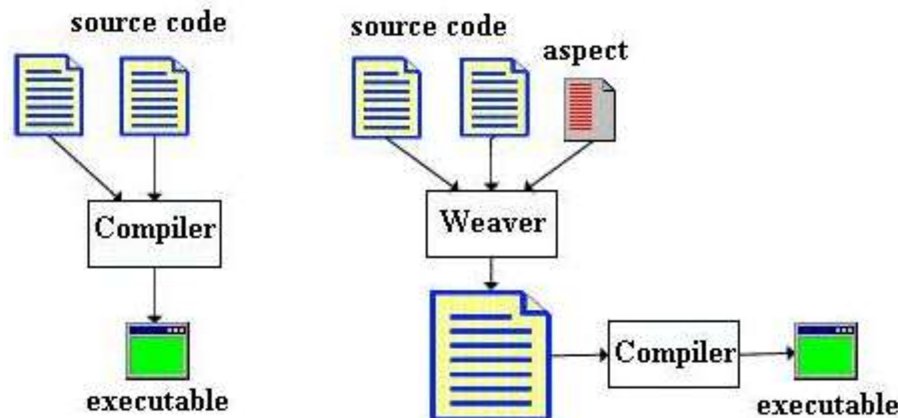

Fig 2: Aspect Weaver

In that way, AOP complements object-oriented programming, not replacing it, by facilitating another type of modularity that pulls together the widespread implementation of a crosscutting concern into a single unit. These units are termed aspects and hence the name aspect oriented programming.

5. CONCLUSIONS AND SUGGESTIONS

AOP is not about writing macros or inserting code at some given line number. Rather, it is about applying certain actions when a specifiable behaviour happens. Thus, mechanisms for aspect orientation rest on three pillars:

- a model of the behaviours that can be recognized and exploited (the joinpoints),
- a means of characterizing a subset of these possible behaviours (the ability to define pointcuts),
- a means of implementing the behaviour defined in the aspects at the place and at the time that the expected behaviour defined in the pointcuts happens (the weaving of the advice).

These three pillars are the "*three critical elements that AO languages have*".

The components need not be aware of the effect of the aspects to which they are subject. In particular, components may be prepared so as to be subjects to aspects, for example via annotations or refactorings, but should not be prepared in a manner that would couple them tightly with the behaviour of any potential aspect to which they might be subject. This notion of *obliviousness* is one of the main assets of AOP for improving the flexibility of software development. This means that, in some cases, the integration of certain aspects into a final version of the code is optional. However, failure to integrate some aspects might completely change the behaviour of the application concerned. For example, an aspect that would check the consistency of some data may be necessary to prevent faults, whereas an aspect that would be used by the programmer for debugging rarely needs to be integrated into the final version of a project.

Reasoning about aspects is still an open problem. Filman and Friedman proposed that "*better AOP systems [should be] more oblivious; they minimize the degree to which programmers (particularly the programmers of the primary functionality) have to change their behavior to realize the benefits of AOP*" [10]. However, full obliviousness has proven to be difficult to achieve in practice. Decoupling crosscutting concerns from the base system gives benefits in term of readability, but full obliviousness can prevent the programmer of the advised units from knowing what will happen when these units are utilised, and in most cases will not be free of undesired side-effects. More recently, Kiczales and Mezini [9] proposed a different way of reasoning about aspects, in which "*aspects cut new interfaces through the primary decomposition of a system. This implies that in the presence of aspects, the complete interface of a module can only be determined once the complete configuration of modules in the system is known. While this may seem anti-modular, it is an inherent property of crosscutting concerns*".

This paper presents that aspects change the concepts of modules as they are used in procedural and object-oriented languages, but provide the ability to view and to reason about cross-sections of the system. The paper also addresses the problems caused by crosscutting concerns as follows: concerns that can be modularized well using the given decomposition mechanisms of a programming language (a.k.a. host programming language) are implemented using these mechanisms. All other concerns that crosscut the implementation of other concerns are implemented as so-called aspects.

It has been seen that an aspect is a kind of module that encapsulates the implementation of a crosscutting concern. It enables code that is associated with one crosscutting concern to be encapsulated into one module, thereby eliminating code scattering and tangling. Moreover, aspects can affect multiple other concerns via one piece of code, thereby avoiding code replication.

Finally we have a brief discussion of aspect weaver which merges the separate aspects of a program and the remaining program elements at predefined join points. This process is called aspect weaving. Join points can be syntactical elements of a program, e.g., a class declaration, or events in the dynamic execution of the program, e.g., a call to a method in the control flow of another method. In further work it can be explored if and how maximizing the use of AOP mechanisms affects the program structure and modularity. It definitely shows promise as a technique for building more reliable software. We hope that programmers will be able to use the technology to encapsulate all sorts of robustness features that were previously difficult to abstract. For example, we see security as an area where aspect oriented programming could greatly ease the burden of the developers. An investigation can be made to find the most superior mechanisms with respect to modularity and understandability. This demands to explore where aspects are useful in expressing homogeneous and advanced dynamic crosscuts.

References

[1] Y. Baldwin and K. B. Clark, "Modularity in the Design of Complex Engineered Systems," in *Complex Engineered Systems: Science Meets Technology*, A. Minai, D. Braha, and Y. B. Yam, Eds. N.Y.: Springer-Verlag, 2006.

[2] T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming: Introduction. Communications of the ACM (CACM), 44(10):29–32, 2001.

[3] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proceedings of International Conference onObjects, Components, Architectures, Services, and Applications for aNetworked World (NetObjectDays)*, 2002.

[4] S. Apel, T. Leich, and G. Saake. Aspect Refinement and Bounded Quantification in Incremental Designs. In *Proceedings of Asia-Pacific Software Engineering Conference (APSEC)*, 2005.

[5] JONATHAN ALDRICH. Open modules: Modular reasoning about advice. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'05), Lecture Notes in Computer Science*, volume 3586, pages 144–168, 2005.

[6] G. Kiczales. Radical Research In Modularity: Aspect-Oriented Programming and Other Ideas. In Keynote of the International Software Product Line Conference (SPLC). IEEE Computer Society, 2006. http://www.sei.cmu.edu/splc2006/splc_kiczales.pdf.

[7] D. S. Dantas and D.Walker. Harmless Advice. In Proceedings of the International Symposium on Principles of Programming Languages (POPL), pages 383–396. ACM Press, 2006.

[8] S. Apel et al. Aspect Refinement. Technical Report 10, Department of Computer Science, University of Magdeburg, Germany, 2006.

[9] GREGOR KICZALES AND MIRA MEZINI. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[10] R. FILMAN AND D. FRIEDMAN. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, 2000.