Aspect Oriented Programming

Gregor Kiczales John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videria Lopes, Chris Maeda, Anurag Mendhekar
Xerox Palo Alto Research Center

Abstract

To date, the primary idea for organizing software systems has been to break the system down into modular units such as subroutines, procedures, objects, clients and servers etc. We note that all of these correspond relatively directly to blocks of executable code. But many issues of concern to programmers don't cleanly follow these modularity boundaries—they don't "fit" naturally into these abstractions.

We propose a new programming paradigm, called Aspect-Oriented Programming, that allows programmers to express each of the different issues they want to program in an appropriately natural form. A special kind of compiler called an Aspect Weaver™ then automatically combins those separate descriptions into a final executable form.

By enabling engineers to reason and program using the natural aspects of concern for a system, even when those cross-cut both each other and the resulting executable code, we believe that Aspect-Oriented Programming will make it possible to program future extremely complex systems, as well as making it easier to program a number of more near term (even present and past) systems.

1 Introduction

To date, the primary idea for organizing software systems has been to break the system down into modular units such as subroutines, procedures, objects, clients and servers etc. We note that all of these correspond relatively directly to blocks of executable code.

But consider an extremely complex system, such as the "Smart Dust" some MEMS researchers have proposed [1]. Briefly speaking, this is a collection of thousands of tiny sensor-processor-communication units that can be distributed randomly over a large area to collectively perform some data gathering function. (Other researchers have proposed similar systems, sometimes referred to as "gunk." or "amorphous computing.")

We claim that such a system cannot be built using the traditional style of modularity. What module do the distribution aspects go in? The power consumption aspects? Information combination? Failure handling? Communication strategy? Interaction of changing geometry and topology? These system aspects fundamentally cut across both each other and the final executable code. An act of communication is also inherently distributed, consumes power, may fail and may require synchronization. The various aspects can be thought about relatively separately, but at the implementation level they must be combined together.

But because modules correspond so directly to blocks of executable code, and the different aspects of concern must cross-cut the executable code, the modules themselves would end up being a tangled mess of aspects. We believe this tangling-of-aspects phenomenon is at the heart of much of the complexity in existing software systems. Further, we believe that increasing the scale of modules (or the level of programming languages) won't help without addressing this root cause of the tangling. Instead what is needed is to be able to work with abstractions that correspond more directly to the real aspects of concern than to blocks of executable code.

We are working on a new programming paradigm, Aspect-Oriented Programming (AOP), that does just this. AOP works by allowing programmers to first express each of a system's aspects of concern in a separate and natural form, and then automatically combine those separate descriptions into a final executable form using a tool called an Aspect Weaver™.

By enabling engineers to reason and program using the natural aspects of concern for a system, even when those cross-cut both each other and the resulting executable code, we believe that AOP will make it possible to program systems as complex as Smart Dust and Gunk. It will also allow us to simplify a number of more near term (even present and past) systems.

We believe that a number of projects, new and old, include intuitions similar to those underlying Aspect-Oriented Programming. The contribution of this short paper is the explicit notion of AOP and an initial set of concepts with which to discuss it. We are also presenting, for the first time, systems constructed with AOP as an explicit guiding principle. In the rest of this short position paper we present examples of AOP, a sketch of the conceptual and technical problems we believe must be addressed, and a comparison with related work.

How AOP Decomposes a system

Using AOP, a system is decomposed into its basic functionality and the cross-cutting aspects. (The basic functionality is sometimes called the primary aspect.) The basic functionality and each of the aspects are captured using appropriate special-purpose languages. Then, all of these programs can be ***woven*** together, to produce code that can then be compiled by an ordinary compiler (i.e. a C compiler).

Examples of the kinds of aspects AOP allows programmers to think and program in terms of are shown in the following table:

Domain:	distributed computing	image processing	numerical simulation
Aspects:	what the objects do	image filters	algorithm
	their location	control structure	numerical stability
	communication	memory usage	data structures
	synchronization	sharing	memory locality

An Example

To date, we have developed several prototypes of aspect-oriented programming in the domains shown in the table above. To provide an example of how AOP "looks and feels," we present a small portion of one of these prototypes. Our focus, in this system, has been on AOP support for a wide range of distributed processing applications. The aspectual decomposition we are working with in this domain breaks systems down into several key aspects, including: the basic functionality of objects, the communication strategy when messages are sent across address space boundaries, and synchronization of the threads of activity

We have developed a basic functionality language and appropriate aspect description languages to capture each of these. The basic functionality language is a simplified C++/Java style language. Programs in this language specify what the objects ***do,*** in the familiar style of imperative object-oriented languages. Programs in the communication aspect language can specify what parameters should be copied, and to what extent, when there are method invocations between objects in different address spaces. Programs in the coordination aspect language define sets of methods that are mutually exclusive and/or auto-exclusive; and define pre-conditions on the execution of methods. The code below is the three programs that define a bounded stack in our system.

Basic Functionality

```
class BStack {
 Integer head;
 Element elts[MAX];
 // other variables here

 void! insert (Element e) {
  if (head == MAX) !;
  else elts[head++] = e;
 }
 Element! remove ( ) {
  if (head == 0) !;
  else elts[--head];
```

Communication Aspect

```
interface BStack {
 void! insert (gref Element);
 gref Element! remove ( );
 gref Element! top ( );
 void newClient (copy Client:id);
};
```

Coordination Aspect

```
relax BStack {
 autoex{insert, remove, newClient};
```

```
  }
  Element! top ( ) {
    if (head == 0) !;
    else elts[head];
  }
  void newClient (Client c){
    //some admin code
  }
};
```

```
  mutex {insert, remove};
  mutex {remove, top};
}
```

Even this small example serves to demonstrate the power of aspectual decomposition and AOP. This same functionality, ***with this same degree of efficiency,*** would be a much more complex program written in a non-aspect oriented way. (In fact, our more complete implementation of this example using our AOP prototype is 43 lines of code, whereas written in Java it would be 120 lines of code.) The aspect-oriented program is easier to write, maintain and reason about than the corresponding code in a system broken down into modules that correspond tightly to blocks of executable code, both because it is smaller and because it corresponds more closely to the issues the programmer really wants to think about.

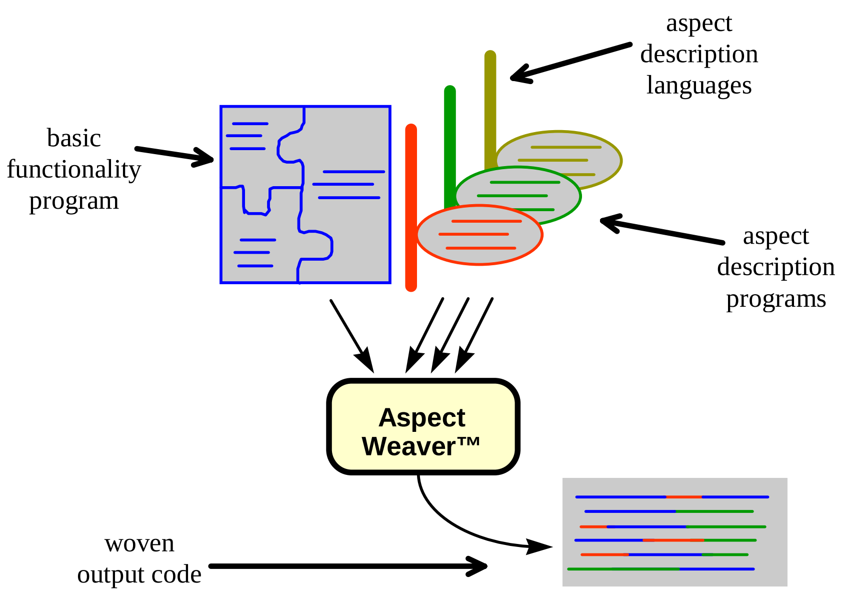

Figure 1 The basic elements of an aspect-oriented programming system. The basic functionality (or primary aspect) is captured using a language that best suits it. Each of the cross-cutting aspects are captured using other appropriately specialized languages. The weaver takes all the programs as input and produces woven output code, which may itself be source code in a language like C.

The reason the non-aspect oriented program is so much longer is that efficiently implementing the communication and synchronization strategies described above in a language like Java would require putting small amounts of code all over the program. This is what we mean when we say that an aspect “cross-cuts” the executable code—even though an issue can be said concisely and locally in a specialized aspect-description language, it must be spread out throughout the executable.

This is what weaving does, it takes the concisely local aspect-description programs and weaves them throughout the resulting executable code. This nature of weaving is capture in Figure 1 which shows how the effect of the different aspect description programs gets spread throughout the woven code. (A color version of this picture is available at the AOP home page, see Summary section.)

Separation with Coordination

Programming in terms of aspectual decomposition requires much more than just decomposing a system aspectually. It requires being able to express those aspects of concern in a way that is precise and that makes the relations among the aspects of concern precise. This is what enables the Aspect Weaver to work, and is also what enables reasoning about the code, debugging the code, and all other aspects of the program life cycle. So, the technical key to making AOP work is to be able to design basic functionality and aspect description languages that on the one hand support a clear natural aspectual decomposition, but on the other hand have a well-defined weaving. The languages used must therefore strike a balance between separating the aspects to achieve clarity and coordinating the aspects to achieve overall coherence.

This balance, and the key to achieving it, can be seen in the example above. Note that all of the programs refer to class and method names. We say that the primary ***join points*** in this system are the method invocations. These three aspect languages identify method invocation in the normal object-oriented programming way—using {class name, method name} pairs. Even though these three aspect description programs talk about how different issues should be handled they share common join points, and a common means of identifying those join points.

This basic structure of aspect languages sharing join points but otherwise addressing different issues appears to be fundamental to AOP. Sharing join points enable the coordination—weaving, partial weaving, reasoning, debugging etc. Otherwise addressing different issues makes them get at different aspects. (If all the terms in the different aspect description languages were the same they wouldn't be getting at ***different*** cross-cutting aspects.)

It is important to note that while in this example the different languages express the join points using the same terms (*i.e.* {class name, method name} pairs) we see no fundamental reason this has to be the case. What seems to be most important is that there be a clearly defined notion of join point that the different languages share. It is also possible for the basic functionality language to share different join points with the different aspect description languages.

In working with our other AOP prototypes we have seen that in each case, the different aspects coordinate with each other and with the overall computation differently. We need to be able to master this range of ways that aspects can interrelate, both in designing aspectual decompositions and in building Aspect Weavers. To approach this problem, we are working with two main themes: the concept of base/meta separation from reflection, which provides an initial intuitive approach for designing such coordination; and developing a solid conceptual analysis of the correspondence relations to provide a more long-term and formalizable foundation.

Reflection

A simple approach to achieving separation and coordination comes from extending the idea of base/meta separation found in reflection and metaobject protocols. The basic intuition underlying computational reflection is that a system can provide a meta-interface, that makes it possible to get at different issues than the base interface, and to get at those issues with a different granularity than the base interface allows.

For example, reflective variants of Lisp provide a primary interface that allows the programmer to define procedures and variables and call them—all the usual semantics of Lisp—and a secondary interface that provides access to such issues as the call sites of a procedure, the places where a variable is referenced, the scope of a closure and so on. Thus the access the meta-interface provides cross-cuts that of the base interface. Using the meta-interface the programmer can concisely refer to things like “all the call sites of a certain procedure,” that are spread all over through the base program.

Base/meta separation can be extended into AOP by designing a base aspect to capture “what to do” and other aspects to capture various aspects of “how to do it.” By having all the auxiliary aspects relate to the base one, their coordination is simplified, and it becomes easier to design an Aspect Weaver that can put them all together.

Analysis of Correspondence

While the base/meta separation principle has proven to be quite powerful, it has three weaknesses that make it insufficient as a complete foundation for AOP: (i) it is informal, (ii) it makes one aspect primary over the others, and (iii) it relates the aspects to each other, rather than relating them to the complete comput ation.

We believe that it is possible to develop a clear conceptual analysis of the correspondence relations in AOP—the ways in which different aspects coordinate with each other and with the final computation. We believe that it will be possible to formalize this analysis into a “correspondence calculus” that could play a role similar to that which the lambda calculus has played in analyzing and building modular systems.

We have developed some initial ideas for this analysis, based on the observation that the structure of connections between the aspects and the executable code is largely one of fan-outs and fan-ins, and that it is the interaction between these that leads to the way in which different aspects cross-cut each other. This analysis framework is nascent, but we have started using it to good benefit in designing our Aspect Weavers.

Other Issues

Having talked about the main technical issue, of being able to design aspect languages that appropriately balance separation and coordination, we now turn to some of the other important issues that must be addressed in order to develop AOP into a working programming paradigm.

Controlling When Computation Happens

An important issue in weaving involves controlling when a computation happens. Many aspect description programs make decisions based on the surrounding computational context into which they are woven. It is vital that decision making not be needlessly repeated each time control passes through the code; it must be once, at weaving time, if possible.

Partial evaluation is one technology for dealing with this issue, since partial evaluation moves computation from run time to partial evaluation time. A key challenge in partial evaluation is determining which computations to move to partial evaluation time, because such motion can have a cost in code duplication. In the case of AOP, though, we can exploit the aspectual decomposition to guide a partial evaluator by indicating information of the form "Do all the computations expressed by a particular aspect at weaving time."

Alternatively, the calculus of correspondence should provide a basis for expressing when a computation should happen. A computation that is done late is one that is repeated—i.e. fans out. A computation that is done early is one that is not repeated—i.e. does not fan out. We can thus express control over when a computation happens in terms of the fan-outs of the calculus of correspondence.

No Smart Compilers

AOP provides a route around what has traditionally been an obstacle to making high-level programming work in performance critical domains. In particular, the AOP paradigm does not require the "smart" compilers that can be so difficult to design and build. In AOP, all the significant implementation decisions, all the smarts, are provided by the programmer using appropriate aspect description languages. The Aspect Weaver's job is integration, rather than inspiration. So, for example, in the example mentioned above, the programmer designs the communication and synchronization ***strategy*** for their system, the weaver only weaves the code that implements it throughout. One way to look at AOP is that it provides the programmer of a specific system with one or more domain-specific languages for functionality and one or more domain specific languages for implementation.

Note that while asking the programmer to explicitly address implementation aspects sounds like it might be a step backwards, our experience to date suggests that in fact it isn't. Consider the example discussed above. It is true that the programmer is addressing implementation, but proper use of AOP means that they are expressing implementation strategy at an appropriately abstract level, through an appropriate aspectual view, with appropriate locality. They are not addressing implementation details, and they are not working directly with the tangled implementation.

This is an important way that AOP differs from traditional high-level language approaches—we ask programmers to address implementation, but to do so in a more appropriate way. Our earlier work on open implementation shows that this can work well [2, 3, 4, 5, 6]. To date, our experience with our demonstration systems suggest it will work well in AOP as well.

We believe this will scale to applications as complex as Smart Dust, where rather than requiring a compiler to automatically envision a communication strategy, the programmers will design that strategy and express it using appropriate aspect languages. This is reasonable since they will already have to have consider communication as part of the system design.

Related Work

A great deal of work, new and old, appears to be based on intuitions similar to those underlying Aspect-Oriented Programming. The literature associated with some of these examples makes an explicit point of using a different kind of decomposition; for others it does not. None of these systems explicitly call out the notion of a new kind of decomposition paradigm or a programming paradigm based on it. But all of them have some important similarities to AOP, in that they are based a decomposition that feels more aspect-oriented than module-oriented.

A important observation is that one can use the notion of aspectual decomposition without necessarily going to the notion of aspect-oriented programming. A system could be aspectually decomposed during design, but still be programmed in a tangled way during implementation. Similarly, a tangled piece of code could be aspectually highlighted by a programming environment to make it somewhat easier to work with than completely tangled code. In looking at related work, we present examples of both of these kinds of systems, as well as some other examples of aspect-oriented programming.

Many design disciplines are based on well-established aspectual decompositions. For example, mechanical engineers use static, dynamic and thermal models of a system as part of designing it. Electrical engineers use various diagrams to reason about their circuits: circuit diagrams, waveform diagrams (timing diagrams), phase diagrams. Each of these models help isolate certain aspects of the circuit. [7]. The coordinate system transforms common in these disciplines can be seen as a kind of shift between aspect description languages. In these domains the weaving together of aspect descriptions is mostly done manually, although certain CAD tools do some of it automat ically.

Software engineers also do this, although often only informally. It is common to hear programmers say something like: “The real issues in this system are X, Y and Z. The reason that line of code is that way is because I need to be sure that ***both*** X and Y are satisfied.” So they have an informal aspectual decomposition that they
“read into” in the tangled code. Most of the object-oriented design methodologies (Objectory, Schlaer-Mellor etc.) provide explicit support for this, by helping

identify aspects that cross-cut each other and track them through the manual process of weaving during the implementation phase.

Program Slicing is a good example of taking cross-cutting views of an existing (tangled) piece of code. Program Slicing is an approach that divides programs into slices dealing with a single programmatic concept: a piece of data, a variable, etc. Tools based on this idea allow the programmer to ask to see all parts of a program that deal with a particular programmatic concept (called a “slice”). Program Slicing is similar to AOP in that slices cross-cut the executable (a.k.a. binary) code. But in Program Slicing the slices are all implemented using the programming language. So while they cross-cut each other, and inherently get at different issues, Program Slicing isn’t providing the level of support for expressing the different issues we’d like to see AOP provide. Program Slicing also differs from AOP in that the emphasis is on the *de*composition—there is no concept of weaving and it does not provide a general way of programming. With respect to AOP, we see program slicing as a powerful tool for helping to untangle existing programs as part of re-engineering them to use AOP.

In the field of hardware design, various projects are attempting to simplify the hardware design process using some form of aspectual decomposition. The Rapid Prototyping of Application Specific Signal Processors (RASSP) program includes many such projects [14]. One of the approaches that the program aims to promote is the separation of communication, computation and control [15] in digital signal processing systems. The Ptolemy project[16], has tried a similar aspectual decomposition for VHDL, where they separate the specification of functional and behavioral issues[16]. Another such project, Graphical Rapid Prototyping Environment (GRAPE II) for rapid prototyping of signal processing and communications systems, uses an aspectual decomposition where the aspects are functionality, partitioning, scheduling and network topology. GRAPE provides an environment in which to combine these aspects into a running DSP pr ogram.

Other examples of AOP

We have found several existing systems that fit our model of aspect-oriented ***programming.*** As mentioned above, since reflection is one good way to get cross-cutting languages that still have the appropriate join points, almost all of the work in reflection can be seen as either leading to the notion of AOP or being an early AOP system. In particular, the last 5 years have seen an increasing interest in using reflection as an engineering tool, and a great deal of experimentation with different meta-level architectures. These different architectures provides different kinds of cross-cutting between the base- and meta-languages, and so provide direct support for work in AOP. A number of researchers in this area have explored the application of reflective techniques to separate issues in distributed computing leading to systems with some of the flavor of the example above [8,9,10].

Composition filters [17] support aspect-oriented programming of certain critical issues in object-oriented programs. They work by intercepting messages that an object receives and "filtering" those messages. Composition filters allow for easy weaving, the filters are composed together and with objects. The work on Composition Filters is a good example of what we are trying to include in the conceptual framework of AOP.

Adaptive Programming is a similar example [12, 13]. In Adaptive Programming, the decomposition is along the lines of data-structure and operations. The operations are written in as data-structure neutral a way as possible, so that the same operations can be reused with different data-structures. Weaving is dealt with explicitly in the way the operations are instantiated for different data-structures.

On a different theme is Synthetix[18], led by Pu and others, which attempts to improve the performance of operating system software by specializing code based on invariants whose truth becomes known only at run-time. The locality of these invariants is non-standard since the physical location of the invariant code is far removed from the physical location of the subject matter of the invariant. Thus the invariants cross-cut our usual notion of the functional decomposition of operating system software.

We believe there are many more examples of systems with AOP-like intuitions and systems that even support aspect-oriented ***programming.*** Just as we are writing this, we are beginning a more extensive search for such systems, which we hope to write up in a future full-length paper.

Summary and Future Directions

We believe the time is right for our field to develop the concept of Aspect-Oriented Programming into a fully-fledged programming paradigm. AOP can help simplify a number of existing systems, and will enable the programming of future extremely complex systems such as amorphous computing. A number of groups are working on AOP type systems, and it appears that an explicit AOP framework will be of value to those efforts just as it has helped us to develop our prototypes. We are maintaining a web site about work in Aspect-Oriented Programming, that includes recent work in AOP and links to such projects, at http://www.parc.xerox.com/aop.

At this time, we see a great many directions that must be pursued in the development of Aspect-Oriented Programming. Some of the most important are:

Quantifying the benefits of AOP. How much can AOP simplify programs? For what kinds of systems does it help the most? How much can this save on lifecycle costs? For what kinds of systems?

Surveying existing work to find projects with intuitions that align with AOP. This should help to clarify our understanding of AOP, and may also help to enhance the existing work.

Developing a theory of programming language semantics that supports the cross-cutting, fan-out and fan-in that are inherently part of aspect-oriented programs. Can we develop a model of how join points work that supports the kinds of AOP systems we want to build? Is there a calculus of correspondence that captures all the kinds of join points we will need to use.

Development of tools support including toolkit support for building weavers, programming environment support for using such weavers, methods for aspectually decomposing a domain etc.

Exploring the notion of library and reusable code in AOP. Will individual aspect descriptions be units of reusable code? If so, how will they be parameterized? If not, what will be?

What are the implications of AOP for software methodologies? How will design tasks be partitioned? Programming tasks? How will aspect-oriented programs be tested? Verified?

Exploring the lifecycle model for AOP. Does the AOP have a different natural software lifecycle? What effect does this have on development and maintenance.

References

1.Berlin, A., *et al.*, *Distributed Information Systems for MEMS, ISAT Study*, . 1995, ISAT.

2.Kiczales, G. *Towards a New Model of Abstraction in Software Engineering.* in *Proceedings of the International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture*. 1992. Tokyo, Japan.

3.Kiczales, G., *Foil for the Workshop on Open Implementation*. 1994: Xerox PARC.

4.Kiczales, G., *Why are Black Boxes so Hard to Reuse?* 1994: Invited Talk, OOPSLA'94.

5.Kiczales, G., *Why Black Boxes are so Hard to Reuse*, 1995.

6.Kiczales, G., *Beyond the Black Box: Open Implementation.* IEEE Software, 1996. **13**(1): p. 8--11.

7.Fisler, K., *Exploiting the Potential of Diagrams in Guiding Hardware Reasoning*, in *Logical Reasoning with Diagrams*, G. Allwein and J. Barwise, Editors. 1996, Oxford University Press.

8.Yokote, Y. *The Apertos Reflective Operating System: The Concept and its Implementation*. in *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications.* 1992.

9.Okamura, H., Y. Ishikawa, and M. Tokoro, *AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework*, in *Proceedings of the International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture.* 1992. p. 36-47.

10. Okamura, H. and Y. Ishikawa, *Object Location Control Using Meta-level Programming*, in *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, T.A. Pareschi, Editor. 1994, Springer-Verlag. p. 299-319.

11. Booch, G., *Object Oriented Design with Applications*. 1994: Benjamin/Cummings.

12. Lopes, C.V. *AP/S++: A Case Study of a MOP for Purposes of Software Evolution*. in *Proceedings of Reflection 96*. 1996. San Francisco.

13. Lieberherr, K.J., I. Silva-Lepe, and C. Xaio, *Adaptive Object-Oriented Programming Using Graph-Based Customization.* Communications of the ACM, 1994. **37**(5): p. 94-101.

14. Agency, A.R.P., *Rapid Prototyping of Application, Specific Signal Processing (RASSP).* .

15. Harr, R.E., *Rapid Prototyping of Application, Specific Signal Processing (RASSP) Overview Presentation*, http://esto.sysplan.com/ESTO/RASSP/Presentation/index.html.

16. Buck, J.T., *et al., Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems.* Int. Journal of Computer Simulation, 1994: p. 155-182.
17. Aksit, M., *et al. Abstracting Object Interactions Using Composition Filters.* in *European Conference on Object-Oriented Programming, Workshop on Object-Based Distributed Programming*. 1993: Springer Verlag.
18. Pu, C., *et al. Optimistic Incremental Specialization: Streamlining a Commercial Operating System*. in *The 15th ACM Symposium on Operating Systems Principles*. 1995. Copper Mountain Resort, Colorado: ACM Press.