Aspect-Oriented Programming

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda,
Cristina Lopes, Jean-Marc Loingtier and John Irwin

Xerox Palo Alto Research Center*

We have found many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement. This forces the implementation of those design decisions to be scattered throughout the code, resulting in "tangled" code that is excessively difficult to develop and maintain. We present an analysis of why certain design decisions have been so difficult to clearly capture in actual code. We call the properties these decisions address *aspects*, and show that the reason they have been hard to capture is that they *cross-cut* the system's basic functionality. We present the basis for a new programming technique, called aspect-oriented programming, that makes it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code. The discussion is rooted in systems we have built using aspect-oriented programming.

1. Introduction

Object-oriented programming (OOP) has been presented as a technology that can fundamentally aid software engineering, because the underlying object model provides a better fit with real domain problems. But we have found many programming problems where OOP techniques are not sufficient to clearly capture all the important design decisions the program must implement. Instead, it seems that there are some programming problems that fit neither the OOP approach nor the procedural approach it replaces.

This paper reports on our work developing programming techniques that make it possible to clearly express those programs that OOP (and POP) fail to support. We present an analysis of why some design decisions have been so difficult to cleanly capture in actual code. We call the issues these decisions address *aspects*, and show that the reason they have been hard to capture is that they *cross-cut* the system's basic functionality. We present the basis for a new programming technique, called aspect-oriented programming (AOP), that makes

* 3333 Coyote Hill Road, Palo Alto, CA 94304, USA. gregor@parc.xerox.com

it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code.
We think of the current state of AOP research as analogous to that of OOP 20 years ago. The basic concepts are beginning to take form, and an expanding group of researchers are using them in their work [1, 4, 13, 28]. Furthermore, while AOP qua AOP is a new idea, there are existing systems that have AOP-like properties. The contribution of this paper is an analysis of the problems AOP is intended to solve, as well as an initial set of terms and concepts that support explicit AOP-based system design.

The paper presents AOP in an example-driven way—the generalizations and definitions are all derived from examples, rather than presented in advance. Section 3 uses a medium-scale example to present the aspect-tangling problem AOP solves; the section culminates with a definition of the term aspect. Section 4 presents several more small examples of aspects. Sections 5 and 6 each provide an example of a complete AOP system. The remaining sections present future work, related work and conclusions.

2. Background Assumptions

This section outlines important assumptions about the relationship between programming languages and software design processes that underlie the rest of the paper.

Software design processes and programming languages exist in a mutually supporting relationship. Design processes break a system down into smaller and smaller units. Programming languages provide mechanisms that allow the programmer to define abstractions of system sub-units, and then compose those abstractions in different ways to produce the overall system. A design process and a programming language work well together when the programming language provides abstraction and composition mechanisms that cleanly support the kinds of units the design process breaks the system into.

From this perspective, many existing programming languages, including object-oriented languages, procedural languages and functional languages, can be seen as having a common root in that their key abstraction and composition mechanisms are all rooted in some form of generalized procedure. For the purpose of this paper we will refer to these as generalized-procedure (GP) languages. (This is not to say that we are ignorant of the many important advantages of OOP languages! It is only to say that for the purposes of the discussion in this paper, it is simpler to focus on what is common across all GP languages.)

The design methods that have evolved to work with GP languages tend to break systems down into units of behavior or function. This style has been

called functional decomposition [25-27].[1] The exact nature of the decomposition differs between the language paradigms of course, but each unit is encapsulated in a procedure/function/object, and in each case, it feels comfortable to talk about what is encapsulated as a *functional* unit of the overall system. This last point may be so familiar that it feels somewhat redundant. But it is important that we give it explicit attention now, because in the course of this paper will be considering units of system decomposition that are not functional.

3. What Are Aspects?

To better understand the origins of tangling problems, and how AOP works to solve them, this section is organized around a detailed example, that is based on a real application we have been working with [18, 22]. There are three implementations of the real application: easy to understand but inefficient, efficient but difficult to understand, and an AOP-based implementation that is both easy to understand and efficient. The presentation here will be based on three analogous but simplified implementations.

Consider the implementation of a black-and-white image processing system, in which the desired domain model is one of images passing through a series of filters to produce some desired output. Assume that important goals for the system are that it be easy to develop and maintain, and that it make efficient use of memory. The former because of the need to quickly develop bug-free enhancements to the system. The latter because the images are large, so that in order for the system to be efficient, it must minimize both memory references and overall storage requirements.

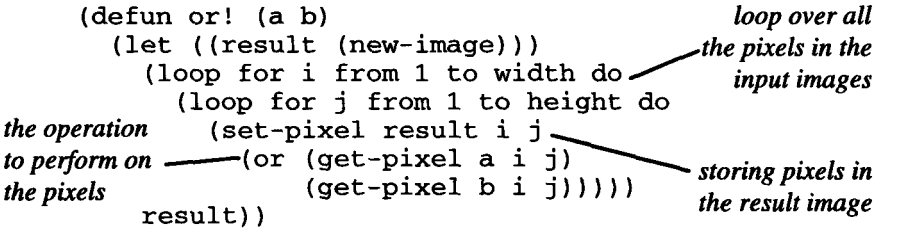
3.1 Basic Functionality

Achieving the first goal is relatively easy. Good old-fashioned procedural programming can be used to implement the system clearly, concisely, and in good alignment with the domain model. In such an approach the filters can be defined as procedures that take several input images and produce a single output image. A set of primitive procedures would implement the basic filters, and higher level filters would be defined in terms of the primitive ones. For example, a primitive `or!` filter, which takes two images and returns their pixelwise logical or, might be implemented as:[2]

---

[1] In some communities this term connotes the use of functional programming languages (i.e. side-effect free functions), but we do not use the term in that sense.

[2] We have chosen Common Lisp syntax for this presentation, but this could be written fairly easily in any other Algol-like language.



Starting from `or!` and other primitive filters, the programmer could work up to the definition of a filter that selects just those black pixels on a horizontal edge, returning a new image consisting of just those boundary pixels.

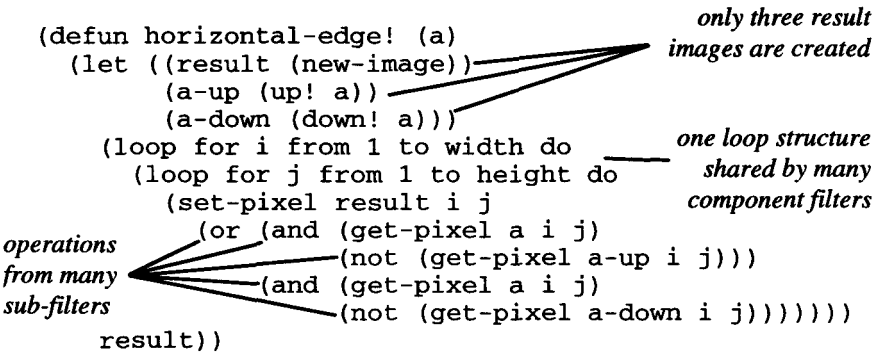
functionality	**implementation**
pixelwise logical operations	written using loop primitive as above
shift image up, down	written using loop primitive; slightly different loop structure
difference of two images	`(defun remove! (a b) (and! a (not! b)))`
pixels at top edge of a region	`(defun top-edge! (a) (remove! a (down! a)))`
pixels at bottom edge of a region	`(defun bottom-edge! (a) (remove! a (up! a)))`
horizontal edge pixels	`(defun horizontal-edge! (a) (or! (top-edge! a) (bottom-edge! a)))`

Note that only the primitive filters deal explicitly with looping over the pixels in the images. The higher level filters, such as `horizontal-edge!`, are expressed clearly in terms of primitive ones. The resulting code is easy to read, reason about, debug, and extend—in short, it meets the first goal.

3.2 Optimizing Memory Usage

But this simple implementation doesn't address the second goal of optimizing memory usage. When each procedure is called, it loops over a number of input images and produces a new output image. Output images are created frequently, often existing only briefly before they are consumed by some other loop. This results in excessively frequent memory references and storage allocation, which in turn leads to cache misses, page faults, and terrible performance.

The familiar solution to the problem is to take a more global perspective of the program, map out what intermediate results end up being inputs to what other filters, and then code up a version of the program that fuses loops appropriately to implement the original functionality while creating as few intermediate images as possible. The revised code for `horizontal-edge!` would look something like:




```
(defun horizontal-edge! (a)
  (let ((result (new-image))
        (a-up (up! a))
        (a-down (down! a)))
    (loop for i from 1 to width do
      (loop for j from 1 to height do
        (set-pixel result i j
          (or (and (get-pixel a i j)
                   (not (get-pixel a-up i j)))
              (and (get-pixel a i j)
                   (not (get-pixel a-down i j)))))))
    result))
```

Compared to the original, this code is all tangled up. It incorporates all the different filters that `horizontal-edge!` is defined in terms of, and fuses many, but not all, of their loops together. (The loops for `up!` and `down!` are not fused because those operations have a different looping structure.)[3] In short, revising the code to make more efficient use of memory has destroyed the original clean component structure.

Of course, this is a very simple example, and it is not so difficult to deal with such a small amount of tangled code. But in real programs the complexity due to such tangling quickly expands to become a major obstacle to ease of code development and maintenance. The real system this example was drawn from is an important sub-component of an optical character recognition system. The clean implementation of the real system, similar to the first code shown above, is only 768 lines of code; but the tangled implementation, which does the fusion optimization as well as memoization of intermediate results, compile-time memory allocation and specialized intermediate datastructures, is 35213 lines. The tangled code is extremely difficult to maintain, since small changes to the functionality require mentally untangling and then re-tangling it.

---

[3] Our AOP-based re-implementation of the full application fuses these other loops as well. We chose not to show that code here because it is so tangled that it is distractingly difficult to understand.

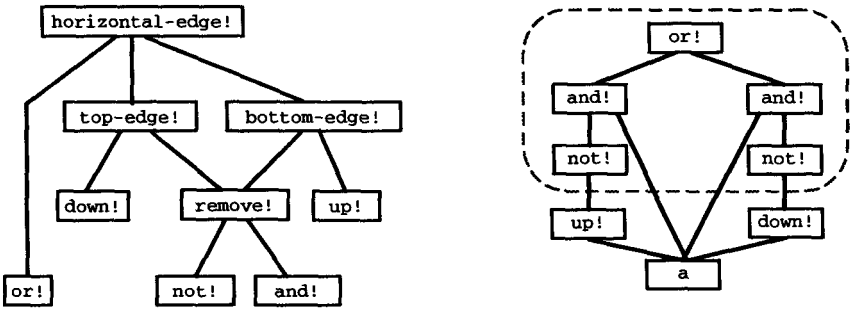


Figure 1: Two different diagrams of the un-optimized `horizontal-edge!` filter. On the left is the functional decomposition, which aligns so directly with the domain model. On the right is a data flow diagram, in which the boxes are the primitive filters and the edges are the data flows between them at runtime. The box labeled a at the bottom is the input image.

3.3 Cross-Cutting

Returning to the example code, Figure 1 provides a different basis for understanding the tangling in it. On the left there is the hierarchical structure of the filtering functionality. On the right there is a data flow diagram for the original, un-optimized version of `horizontal-edge!`. In this diagram, the boxes and lines show the primitive filters and data flow between them. The dashed oval shows the boundary of what is fused into a single loop in the optimized version of `horizontal-edge!`.

Notice that the fusion oval does not incorporate all of `horizontal-edge!` In fact, it doesn't align with any of the hierarchical units on the left. While the two properties being implemented—the functionality and the loop fusion—both originate in the same primitive filters, they must compose differently as filters are composed. The functionality composes hierarchically in the traditional way. But the loop fusion composes by fusing the loops of those primitive filters that have the same loop structure and that are direct neighbors in the data flow graph. Each of these composition rules is easy to understand when looking at its own appropriate picture. But the two composition relationships cut each other so fundamentally that each is very difficult to see in the other's picture.

This cross-cutting phenomena is directly responsible for the tangling in the code. The single composition mechanism the language provides us—procedure calling—is very well suited to building up the un-optimized functional units. But it can't help us compose the functional units and the loop fusion simultaneously, because they follow such different composition rules and yet must co-compose.

This breakdown forces us to combine the properties entirely by hand—that's what happening in the tangled code above.

In general, whenever two properties being programmed must compose differently and yet be coordinated, we say that they *cross-cut* each other. Because GP languages provide only one composition mechanism, the programmer must do the co-composition manually, leading to complexity and tangling in the code.

We can now define two important terms more precisely:

> With respect to a system and its implementation using a GP-based language, a property that must be implemented is:
>
> **A component, if it can be cleanly encapsulated in a generalized procedure** (i.e. object, method, procedure, API). By cleanly, we mean well-localized, and easily accessed and composed as necessary. Components tend to be units of the system's functional decomposition, such as image filters, bank accounts and GUI widgets.
>
> **An aspect, if it can not be cleanly encapsulated in a generalized procedure.** Aspects tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways. Examples of aspects include memory access patterns and synchronization of concurrent objects. (Section 4 provides more examples of aspects.)

Using these terms it is now possible to clearly state the goal of AOP: To support the programmer in cleanly separating components and aspects from each other,[4] by providing mechanisms that make it possible to abstract and compose them to produce the overall system. This is in contrast to GP-based programming, which supports programmers in separating only components from each other by providing mechanisms that make it possible to abstract and compose them to produce the overall system.[5]

---

[4] Components from each other, aspects from each other, and components from aspects.

[5] Our analysis of aspects as system properties that cross-cut components helps explain the persistent popularity of mechanisms like dynamic scoping, catch and throw in otherwise purely GP languages. These mechanisms provide a different composition mechanism, that helps programmers implement certain aspects in their systems.

4. Other Examples of How Aspects Cross-Cut Components

Before going on to the presentation of AOP, and how it solves the problem of aspect tangling in code, this section briefly presents several more examples of aspects and components. For each example in the table below we list an application, a kind of GP language that would do a good job of capturing the component structure of the application, a likely component structure for the application if programmed using that kind of language, and the aspects that would cross-cut that component structure.

application	GP language	components	aspects
image processing	procedural	filters	loop fusion result sharing compile-time memory allocation
digital library	object-oriented	repositories, printers, services	minimizing network traffic synchronization constraints failure handling
matrix algorithms	procedural	linear algebra operations	matrix representation permutation floating point error

Some aspects are so common that they can easily be thought about without reference to any particular domain. One of the best examples is error and failure handling. We are all familiar with the phenomenon that adding good support for failure handling to a simple system prototype ends up requiring many little additions and changes throughout the system. This is because the different dynamic contexts that can lead to a failure, or that bear upon how a failure should be handled, cross-cut the functionality of systems.

Many performance-related issues are aspects, because performance optimizations often exploit information about the execution context that spans components.

5. First Example of AOP

In this section we return to the image processing example, and use it to sketch an AOP-based re-implementation of that application. The presentation is based on a system we have developed, but is simplified somewhat. The complete system is discussed in [22]. The goal of this section is to quickly get the complete structure of an AOP-based implementation on the table, not to fully explain that structure. Section 6 will provide that explanation.

The structure of the AOP-based implementation of an application is analogous to the structure of a GP-based implementation of an application. Whereas a GP-based implementation of an application consists of: (i) a language, (ii) a compiler (or interpreter) for that language, and (iii) a program written in the language that implements the application; the AOP-based implementation of an application consists of: (i.a) a *component language* with which to program the components, (i.b) one or more *aspect languages* with which to program the aspects, (ii) an a*spect weaver* for the combined languages, (iii.a) A *component program*, that implements the components using the component language, and (iii.b) one or more *aspect programs* that implement the aspects using the aspect languages. Just as with GP-based languages, AOP languages and weavers can be designed so that weaving work is delayed until runtime (RT weaving), or done at compile-time (CT weaving).

5.1 The Component Language & Program

In the current example we use one component language and one aspect language. The component language is similar to the procedural language used above, with only minor changes. First, filters are no longer explicitly procedures. Second, the primitive loops are written in a way that makes their loop structure as explicit as possible. Using the new component language the `or!` filter is written as follows:

```
(define-filter or! (a a)
  (pixelwise (a b) (aa bb) (or aa bb)))
```

The `pixelwise` construct is an iterator, which in this case walks through images `a` and `b` in lockstep, binding `aa` and `bb` to the pixel values, and returning a image comprised of the results. Four similar constructs provide the different cases of aggregation, distribution, shifting and combining of pixel values that are needed in this system. Introducing these high-level looping constructs is a critical change that enables the aspect languages to be able to detect, analyze and fuse loops much more easily.

5.2 The Aspect Language & Program

The design of the aspect language used for this application is based on the observation that the dataflow graph in Figure 1 makes it easy to understand the loop fusion required. The aspect language is a simple procedural language that provides simple operations on nodes in the dataflow graph. The aspect program can then straightforwardly look for loops that should be fused, and carry out the fusion required. The following code fragment is part of the core of that aspect program—it handles the fusion case discussed in Section 5. It checks whether two nodes connected by a data flow edge both have a pixelwise loop structure, and if so it fuses them into a single loop that also has a pixelwise structure, and that has the appropriate merging of the inputs, loop variables and body of the two original loops.

```
(cond ((and (eq (loop-shape node) 'pointwise)
            (eq (loop-shape input) 'pointwise))
       (fuse loop input 'pointwise
             :inputs (splice ...)
             :loop-vars (splice ...)
             :body (subst  ...))))
```

Describing the composition rules and fusion structure for the five kinds of loops in the real system requires about a dozen similar clauses about when and how to fuse. This is part of why this system could not be handled by relying on an optimizing compiler to do the appropriate fusion—the program analysis and understanding involved is so significant that compilers cannot be counted upon to do so reliably. (Although many compilers might be able to optimize this particular simple example.) Another complication is the other aspects the real system handles, including sharing of intermediate results and keeping total runtime memory allocation to a fixed limit.

5.3 Weaving

The aspect weaver accepts the component and aspect programs as input, and emits a C program as output. This work proceeds in three distinct phases, as illustrated in Figure 2.

In phase 1 the weaver uses unfolding as a technique for generating a data flow graph from the component program. In this graph, the nodes represent primitive filters, and the edges represent an image flowing from one primitive filter to another. Each node contains a single loop construct. So, for example, the node labeled A contains the following loop construct, where the #<...> refer to the edges coming into the node:

```
(pointwise (#<edge1> #<edge2>) (i1 i2) (or i1 i2))
```

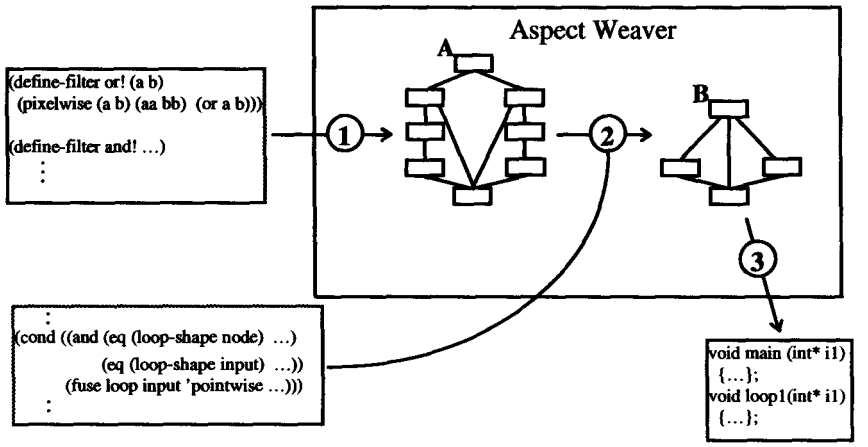


Figure 2: The aspect weaver for image processing applications works in three phases.

In phase 2 the aspect program is run, to edit the graph by collapsing nodes together and adjusting their bodies accordingly. The result is a graph in which some of the loop structures have more primitive pixel operations in them than before phase 2. For example, the node labeled **B**, which corresponds to the fusion of 5 loops from the original graph, has the following as its body:

```
(pointwise (#<edge1> #<edge2> #<edge3>) (i1 i2 i3)
  (or (and (not i1) i2) (and (not i3) i2))))
```

Finally, in phase 3, a simple code generator walks over the fused graph, generating one C function for each loop node, and generating a main function that calls the loop functions in the appropriate order, passing them the appropriate results from prior loops. The code generation is simple because each node contains a single loop construct with a body composed entirely of primitive operations on pixels.

A crucial feature of this system is that the weaver is not a "smart" compiler, which can be so difficult to design and build. By using AOP, we have arranged for all the significant implementation strategy decisions—all the actual smarts—to be provided by the programmer, using the appropriate aspect languages. The weaver's job is integration, rather than inspiration.[6]

---

[6] While asking the programmer to explicitly address implementation aspects sounds like it might be a step backwards, our experience with work on open implementation suggests that in fact it isn't [9, 10, 12, 17] While the programmer is addressing implementation in the memory aspect, proper use of AOP means that they are expressing implementation strategy at an appropriately abstract level, through an appropriate aspect language, with appropriate locality. They are not addressing implementation details, and they are not working directly with the tangled implementation. In evaluating the AOP-based implementation it is important to compare it with both the naïve inefficient implementation and the complex efficient implementation.

5.4 Results

The real system is somewhat more complex of course. For one thing, there are two additional aspect programs, one of which handles sharing of common sub-computations, and one of which ensures that the minimum possible number of images are allocated at any one time. In this system, all three of the aspect programs are written in the same aspect language.

In this example, the AOP based re-implementation has met the original design goals—the application code is easy to reason about, develop and maintain, while at the same time being highly efficient. It is easy for the programmer to understand the components and how they compose. It is easy for the programmer to understand the aspects, and how they compose. It is easy for the programmer to understand the effect of the aspect programs on the total output code. Changes in either the filter components or the fusion aspect are easily propagated to the whole system by simply re-weaving. What isn't easy is for the programmer to generate the details of the output code. The power of the AOP approach is that the weaver handles these details, instead of the programmer having to do the tangling manually.

Our AOP based re-implementation of the application is 1039 lines of code, including the component program and all three aspect programs. The aspect weaver itself, including a reusable code generation component is 3520 lines (the true kernel of the weaver is 1959 lines). The performance of the re-implementation is comparable to a 35213 line manually tangled version (the time efficiency is worse and the space efficiency is better).[7]

As with many other software engineering projects, it is extremely difficult to quantify the benefits of using AOP without a large experimental study, involving multiple programmers using both AOP and traditional techniques to develop and maintain different applications [6, 21, 36]. Such a study has been beyond the scope of our work to date, although we hope to do one in the future. In the meantime, we have developed one initial measure of the degree to which applying AOP techniques can simplify an application. This measure compares a GP-based implementation of an application to an AOP-based implementation of the same application. It measures the degree to which the aspects are more concisely coded in the AOP-based implementation than in a non-AOP based implementation. The general equation for this measure, as well as the numbers for this particular application are as follows:

---

[7] Our current code generator doesn't use packed datastructures, this results in a factor of 4 performance penalty between the hand-optimized implementation and the aspect-oriented implementation. The aspect-oriented implementation is nonetheless over 100 times faster than the naive implementation.

$$\text{reduction in bloat due to tangling} = \frac{\text{tangled code size - component program size}}{\text{sum of aspect program sizes}} = \frac{35213 - 756}{352} = 98$$

In this metric, any number greater than 1 indicates a positive outcome of applying AOP. This application represents an extremely large gain from using AOP, in other applications we have developed the gain ranges from 2 to this number [8, 14, 22]. It could be said that the size of the weaver itself should be included in the sum in the denominator. The point is debatable, since the weaver is usable by any number of similar image processing applications, not just the table recognizer. But we note that even with the entire weaver included, this metric evaluates to 9.

Any single metric has somewhat limited utility. We believe that this one is useful in this case because on the other important grounds of performance, the AOP-based implementation of the application is comparable to the non-AOP based implementation. Section 7 presents some of the requirements we have identified for quantitative measures of AOP utility.

6. Second Example of AOP

This section uses a second example of an AOP-based system to elaborate on component language design, aspect language design and weaving. Once again, the example is a simplified version of a real system we are developing, which is described in [14]. The example comes from the document processing domain where we wanted to implement a distributed digital library that stores documents in many forms and provides a wide range of operations on those documents. The component language, aspect languages and aspect weaver presented in this section are more general-purpose in nature than the highly domain-specific example in the previous section.

The functionality of this system is well captured using an object-oriented model. In such an approach the objects are documents, repositories, different printable forms for the documents (pdf, ps, rip...), printers, servers etc. There are several aspects of concern, including:

- Communication, by which we mean controlling the amount of network bandwith the application uses by being careful about which objects and sub-objects get copied in remote method calls. For example, we want to be sure that when a book object is included in a remote method invocation, the different printed representations of the book aren't sent across the wire unless they will be needed by the receiving method.
- Coordination constraints, by which we mean the synchronization rules required to ensure that the component program behaves correctly in the face of multiple threads of control.

- Failure handling, by which we mean handling the many different forms of failure that can arise in a distributed system in an appropriately context-sensitive way.

For now, we will continue with just the communication aspect. Handling both communication and coordination using AOP is discussed in [14]. Failure handling using AOP is a future research goal.

6.1 The Component Language & Program

Designing an AOP system involves understanding what must go into the component language, what must go into the aspect languages, and what must be shared among the languages. The component language must allow the programmer to write component programs that implement the system's functionality, while at the same time ensuring that those programs don't pre-empt anything the aspect programs need to control. The aspect languages must support implementation of the desired aspects, in a natural and concise way. The component and aspect languages will have different abstraction and composition mechanisms, but they must also have some common terms, these are what makes it possible for the weaver to co-compose the different kinds of programs.

To keep the common terms from becoming points of contention, the aspect languages must address different issues than the component languages. In the image processing system, replacing low-level loops with the higher-level looping primitives is an example of ensuring that component programs don't pre-empt aspect programs. This change makes it easier for the aspect programs to detect and implement opportunities for loop fusion.

In this example, component programs must implement elements such as books, repositories, and printers. In order to allow the communication aspect program to handle communication, component programs must avoid doing so. In this case Java™ serves quite well as the component language. It provides an object model that implements the appropriate components, and avoids addressing the communication aspect.[8] So, using Java as our component language, the definition of two simple classes, books and repositories of books, look like:

---

[8] [14] explains that in order to support the coordination aspect language, some lower-level synchronization features must be removed from Java before it can be used as the component language. These are the keyword `synchronized`, and the methods `wait`, `notify` and `notifyAll`.

```
public class Book {
  String title, author;
  int isbn;
  OCR ocr;
  PDF pdf;
  Postscript ps;
  RIP rip;

  public String get_title()
{
    return title;
  }
  public String get_author()
{
    return author;
  }
public int get_isbn() {
    return isbn;
  }
}
```

```
public class Repository {
  private Book     books[];
  private int      nbooks = 0;

  public Repository (int dbsize)
  {
    books = new Book[dbsize];
  }
  public void register (Book b)
  {
    books[nbooks++] = b;
  }
  public void unregister(Book b)
  {   ...  }
  public Book lookup (String s)
  {   ...  }
}
```

6.2 The Aspect Language & Program

Communication aspect programs would like to be able to control the amount of copying of arguments that takes place when there is a remote method invocation. To do this, the aspect language must effectively allow them to step into the implementation of method invocation, to detect whether it is local or remote, and to implement the appropriate amount of copying in each case.

One way to do this is to provide runtime reflective access to method invocation. As has been shown in [7, 23, 35, 37] such reflective access can be used to control the communication aspect of a distributed object system. But this kind of reflective access is so powerful that it can be dangerous or difficult to use. So in this case we have chosen to provide a higher-level aspect language, that is more tailored to the specific aspect of controlling copying in remote method invocations.

The communication aspect language we have designed allows the programmer to explicitly describe how much of an object should be copied when it is passed as an argument in a remote method invocation. Using this language, the following fragment of the communication aspect program says that when books are registered with a repository, all of their sub-objects should be copied; when they are de-registered or returned as the result of a lookup, only the ISBN number is copied. The rest of the book, including large sub-objects such as the printable representations, is not copied unless it is needed at some later time.

```
remote Repository {
  void register (Book);
  void unregister (Book: copy isbn);
  Book: copy isbn lookup(String);
}
```

6.3 Aspect Weaver

Aspect weavers must process the component and aspect languages, co-composing them properly to produce the desired total system operation. Essential to the function of the aspect weaver is the concept of *join points*, which are those elements of the component language semantics that the aspect programs coordinate with.

In the image processing example, the join points are the data flows of the component program. In this distributed objects example, the join points are the runtime method invocations in the component program. These two examples serve to illustrate an important point about join points—they are not necessarily explicit constructs in the component language. Rather, like nodes in the dataflow graph and runtime method invocations they are clear, but perhaps implicit, elements of the component program's semantics.

Aspect weavers work by generating a *join point representation* of the component program, and then executing (or compiling) the aspect programs with respect to it. In the digital library example, the join-point representation includes information about dynamic method invocations such as the concrete classes of the arguments and their location. The join point representation can be generated at runtime using a reflective runtime for the component language. In this approach, the aspect language is implemented as a meta-program, called at each method invocation, which uses the join point information and the aspect program, to know how to appropriately marshal the arguments.[9] Thus the higher-level aspect language we have designed is implemented on top of a lower level one, as often happens in GP languages.

In the image processing application, the join point representation is quite simple. It is just the data flow graph, operations to access the body of nodes, and operations to edit the graph.

---

[9] In our actual system we use compile-time reflective techniques, so that no interpretive overhead is incurred at runtime.

7. Open Issues

As an *explicit* approach to programming, AOP is a young idea. Our work to date has been primarily focused on designing and implementing aspect-oriented programming languages, and using those languages to develop prototype applications. This programming-centric initial focus has been natural, and it parallels the early development of OOP. But there is a great deal of work still to be done to assess the overall utility of AOP, to better understand its relation to existing ideas, and to further develop it so that it can be useful for a wide range of users.

One important goal is quantitative assessment of the utility of AOP. How much does it help in the development of real-world applications? How much does it help with maintenance? Can we develop measures of which applications it will be more or less useful for? This is a difficult problem, for all the same reasons that quantitative assessment of the value of OOP has been difficult, but we believe that it is important to begin work on this, given that it will take time to get solid results.

We also believe it is important to begin a systematic study to find existing systems that have AOP-like elements in their design. We see this as a way to quickly accelerate development of the AOP ideas, by providing a way to get rough empirical evidence without having to build large new systems from the ground up.

Another important area for exploration is the space of different kinds of component and aspect language designs. Can we develop a collection of component and aspect languages that can be plugged together in different ways for different applications? Can we use meta-level frameworks [2, 3, 20, 38] to build such a collection?

What theoretical support can be developed for AOP? What kinds of theories can best describe the interaction between aspects and components and how they must be woven? Can such theories support development of a practical weaving toolkit?

What about the analysis and design process? What are good design principles for aspectual decomposition? What are good "module" structures for aspect programs? How can we train people to identify aspects? Clearly separate them? Write aspect programs? Debug AOP systems? Document AOP systems?

Another important area of exploration is the integration of AOP with existing approaches, methods, tools and development processes. As the examples in this paper show, AOP can be used as an improvement to existing techniques. To fulfill this promise it must developed it a way that integrates well with those techniques.

8. Related Work

In this section we give a brief survey of work related to ours. We start with work that is more closely related and proceed out to work that is less closely related.

8.1 Work Explicitly Connected to AOP

Several other groups have begun to explicitly consider their work in AOP terms. These include:

- Mehmet Aksit et. al., at the University of Twente, have developed the composition filters object model, which provides control over messages received and sent by an object [1]. In their work, the component language is a traditional OOP, the composition filters mechanism provides an aspect language that can be used to control a number of aspects including synchronization and communication. Most of the weaving happens at runtime; the join points are the dynamic message sends and receives arriving at an object.
- Calton Pu et. al. at the Oregon Graduate Institute, in their work on Synthetix, are developing high performance, high portability and high adaptiveness OS kernels [19, 28]. In their work, the components are familiar functional elements of OS kernels. The aspects are primarily optimizations based on invariants that relate to how a service is being used. Their weaver technology uses partial evaluation to effectively specialize the kernel code for particular use cases. Their code is structured to expose as join points those places where an invariant becomes or ceases to be true.
- Karl Lieberherr et. al., at Northeastern University are developing techniques that make object-oriented programs more reusable and less brittle in the face of common program evolution tasks [13, 15, 31]. In their work, the component languages are existing OOPs like C++ and Java. Succinct traversal specifications [13] and context objects [31] provide aspect languages that can be used to address a variety of cross-cutting issues. Weaving of aspect programs that use succinct traversal specification is compile-time oriented, the join point representation is, roughly speaking, the class graph. Weaving of aspect programs that use context objects is more runtime oriented, the join points are the dynamic method and function calls.

8.2 Reflection and Metaobject Protocols

Aspect-oriented programming has a deep connection with work in computational reflection and metaobject protocols [11, 20, 24, 32, 34, 38]. A reflective system provides a base language and (one or more) meta-languages that provide control over the base language's semantics and implementation. The meta languages provide views of the computation that no one base language component could ever see, such as the entire execution stack, or all calls to objects of a given class. Thus, they cross-cut the base level computation. In AOP terms, meta-languages are lower-level aspect languages whose join points are the "hooks" that the reflective system provides. AOP is a goal, for which reflection is one powerful tool.

We have exploited this connection to great advantage in our work on AOP. When prototyping AOP systems we often start by developing simple metaobject protocols for the component language, and then prototype imperative aspect programs using them. Later, once we have a good sense of what the aspect programs need to do, we develop more explicit aspect language support for them.

The connection is particularly evident in section 6, where the aspect languages we provided could have been layered on top of a reflective architecture. Similarly, the loop fusion aspect described in Section 5 can be implemented, with some degree of efficiency, using the method combination facility in the CLOS metaobject protocol [11, 33]. This connection is also evident in the work mentioned in Section 8.1; both the Demeter work and the composition filters work have been described as being reflective facilities [16].

8.3 Program Transformation

The goal of work in program transformation is similar to that of AOP. They want to be able to write correct programs in a higher-level language, and then mechanically transform those program into ones with identical behavior, but more efficient performance. In this style of programming, some of the properties the programmer wants to implement are written in an initial program. Other properties are added by passing that initial program through various transformation programs. This separation is similar in spirit to the component/aspect program separation.

But the notion of component and aspect are new to AOP. These terms provide additional value in system design. Also, while some transformations are aspectual in nature, others are not. Transformation programs tend to operate in terms of the syntax of the program being transformed. If other join points are desired, it is the responsibility of the transformation program to somehow manifest them. Thus, while it is possible to layer some kinds of aspect programs on top of a program transformation substrate, that is a separate piece of implementation work.

We would like to do a systematic analysis of the transformations developed by this community, to see which of them can be used for providing different kinds of aspect languages.

8.4 Subjective Programming

A natural question to ask is whether subjective programming [5] is AOP or vice versa. We believe that AOP and subjective programming are different in important ways. Analogously to the way object-oriented programming supports automatic selection among methods for the same message from different classes, subjective programming supports automatic combination of methods for a given message from different subjects. In both cases, the methods involved are components in the AOP sense, since they can be well localized in a generalized pro-

cedure. It is even possible to program in either an object-oriented style or a subjective style on top of an ordinary procedural language, without significant tangling. The same is not true of AOP. Thus, while the aspects of AOP tend to be about properties that affect the performance or semantics of components, the subjects of subjective programming tend to be additional features added onto other subjects. We believe that subjective programming is complementary to, and compatible with, AOP.

8.5 Other Engineering Disciplines

Many other engineering disciplines are based on well-established aspectual decompositions. For example, mechanical engineers use static, dynamic and thermal models of the same system as part of designing it. The differing models cross-cut each other in that the different properties of a system compose differently. Similarly, some software development tools explicitly support particular aspectual decomposition: tools for OMT [29, 30] methods let programmers draw different pictures of how objects should work.

9. Conclusions

We have traced the complexity in some existing code to a fundamental difference in the kinds of properties that are being implemented. Components are properties of a system, for which the implementation can be cleanly encapsulated in a generalized procedure. Aspects are properties for which the implementation cannot be cleanly encapsulated in a generalized procedure. Aspects and cross-cut components cross-cut each other in a system's implementation.

Based on this analysis, we have been able to develop aspect-oriented programming technology that supports clean abstraction and composition of both components and aspects. The key difference between AOP and other approaches is that AOP provides component and aspect languages with different abstraction and composition mechanisms. A special language processor called an aspect weaver is used to coordinate the co-composition of the aspects and components.

We have had good success working with AOP in several testbed applications. The AOP conceptual framework has helped us to design the systems, and the AOP-based implementations have proven to be easier to develop and maintain, while being comparably efficient to much more complex code written using traditional techniques.

Acknowledgments

Thanks to Karl Lieberherr, Carine Lucas, Gail Murphy and Bedir Tekinerdogan who generously provided extensive comments on earlier drafts of the paper, and to Andy Berlin, Geoff Chase, Patrick Cheung, John Gilbert, Arthur Lee, Calton Pu, Alex Silverman, Marvin Theimer and Mark Yim with whom we have had many discussions about AOP.

Thanks also to all the attendees of the AOP Friends Meetings, with whom we spent an enjoyable two days discussing AOP and related ideas: Mehmet Aksit, Lodewick Bergmans, Pierre Cointe, William Harrison, Jacques Malenfant, Satoshi Matsuoka, Kim Mens, Harold Ossher, Calton Pu, Ian Simmonds, Perri Tarr, Bedir Tekinerdogan, Mark Skipper, and Patrick Steyaert

Bibliography

1. Aksit M., Wakita K., et al., *Abstracting object interactions using composition filters*, in proc. *ECOOP'93 Workshop on Object-Based Distributed Programming*, pp. 152-184, 1993.
2. Bobrow D. G., DeMichiel L. G., et al., *Common Lisp Object System Specification*, in *SIGPLAN Notices*, vol. 23, 1988.
3. Chiba S., *A Metaobject Protocol for C++*, in proc. *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 95)*, Austin, 1995.
4. Consel C., *Program Adaptation based on Program Transformation*, in proc. *ACM Workshop on Strategic Directions in Computing Research*, 1996.
5. Harrison W. and Ossher H., *Subject-oriented programming (a critique of pure objects)*, in proc. *Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pp. 411--428, Washington D.C., 1993.
6. Henry S. and Kafura D., *Software Structure Metrics Based on Information Flow*, in *IEEE Transactions on Software Engineering*, vol. SE-7: 509--518, 1981.
7. Ichisugi Y., Matsuoka S., et al., *Rbc1: A reflective object-oriented concurrrent language without a run-time kernel*, in proc. *International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture*, pp. 24--35, 1992.
8. Irwin J., Loingtier J.-M., et al., *Aspect-Oriented Programming of Sparse Matrix Code*, Xerox PARC, Palo Alto, CA. Technical report SPL97-007 P9710045, February, 1997.
9. Kiczales G., *Foil for the Workshop on Open Implementation*, Xerox PARC, Web pages, http://www.parc.xerox.com/spl/eca/oi/workshop-94/foil/main.html
10. Kiczales G., *Why are Black Boxes so Hard to Reuse?*, Invited Talk, OOPSLA'94, Video tape, Web pages, http://www.parc.xerox.com/spl/eca/oi/gregor-invite.html

11. Kiczales G., des Rivères J., et al., *The Art of the Metaobject Protocol.* Book published by MIT Press, 1991.
12. Kiczales G., Lamping J., et al., *Open Implementation Design Guidelines*, in proc. *International Conference on Software Engineering*, (Forthcoming), 1997.
13. Lieberherr K. J., Silva-Lepe I., et al., *Adaptive Object-Oriented Programming Using Graph-Based Customization*, in *Communications of the ACM*, vol. 37(5): 94-101, 1994.
14. Lopes C. V. and Kiczales G., *D: A Language Framework for Distributed Programming*, Xerox PARC, Palo Alto, CA. Technical report SPL97-010 P9710047, February, 1997.
15. Lopes C. V. and Lieberherr K., *Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications*, in proc. *European Conference on Object-Oriented Programming*, pp. 81-99, Bologna, Italy, 1994.
16. Lopes C. V. and Lieberherr K., *AP/S++: Case-Study of a MOP for Purposes of Software Evolution*, in proc. *Reflection'96*, pp. 167-184, S. Francisco, CA, 1996.
17. Maeda C., Lee A., et al., *Open Implementation Analysis and Design*, in proc. *Symposium on Software Reuse (To Appear, May 1997)*, 1997.
18. Mahoney J. V., *Functional Visual Routines*, Xerox Palo Alto Research Center, Palo Alto SPL95-069, July 30, 1995, 1995.
19. Massalin H. and Pu C., *Threads and Input/Output in the Synthesis Kernel*, in *Proceedings of the 12th ACM Symposium on Operating Systems Principles* :pp 191-201, 1989.
20. Matsuoka S., Watanabe T., et al., *Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming*, in *European Conference on Object Oriented Programming* :pp 231-250, 1991.
21. McClure C., *A Model for Program Complexity Analysis*, in proc. *3rd International Conference on Software Engineering*, Los Alamitos, CA, 1978.
22. Mendhekar A., Kiczales G., et al., *RG: A Case-Study for Aspect-Oriented Programming*, Xerox PARC, Palo Alto, CA. Technical report SPL97-009 P9710044, February, 1997.
23. Okamura H., Ishikawa Y., et al., *Al-1/d: A distributed programming system with multi-model reflection framework*, in proc. *International Workshop on New Models for Software Architecture '92; Reflection and Meta-Level Architecture*, pp. 36--47, 1992.
24. Okamura H., Ishikawa Y., et al., *Metalevel Decomposition in AL-1/D*, in proc. *International Symposium on Object Technologies for Advanced Software*, pp. 110-127, 1993.
25. Parnas D. L., *Designing Software for Extension and Contraction*, in proc. *3rd International Conference on Software Engineering*, pp. 264-277, 1978.
26. Parnas D. L., *On a 'Buzzword': Hierarchical Structure*, in proc. *IFIP Congress 74*, pp. 336-339, 1974.
27. Parnas D. L., *On the Criteria to be Used in decomposing Systems into Modules*, in *Communications of the ACM*, vol. 15(2), 1972.

28. Pu C., Autrey T., et al., *Optimistic Incremental Specialization: Streamlining a Commercial Operating System*, in proc. *15th ACM Symposium on Operating Systems Principles (SOSP'95)*, 1995.
29. Rational, *Rational Web pages*, Rational Software Corporation, Web pages, http://www.rational.com
30. Rumbaugh J., Blaha M., et al., *Object-Oriented Modeling and Design*. Book published by Prentice Hall, 1991.
31. Seiter L. M., Palsberg J., et al., *Evolution of Object Behavior Using Context Relations*, in proc. *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 46--57, San Francisco, 1996.
32. Smith B. C., *Reflection and Semantics in a Procedural Language LCS Technical Report*, M.I.T., Cambridge, MA, 1982.
33. Steele G. L., *Common LISP: The Language, 2nd Edition*. Book published by Digital Press, 1990.
34. Wand M. and Friedman D. P., *The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower*, in *Proceedings of the ACM Conference on LISP and Functional Programming* :pp 298-307. ACM, 1986.
35. Watanabe T. and Yonezawa A., *Reflection in an object-oriented concurrent language*, in proc. *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 88)*, pp. 306--315, San Diego, CA, 1988.
36. Yau S. and Collofello J., *Some Stability Measures for Software Maintenance*, in *tse*, vol. SE-6: 545--552, 1980.
37. Yokote Y., *The Apertos Reflective Operating System: The Concept and its Implementation*, in proc. *Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 1992.
38. Yonezawa A. and Watanabe T., *An Introduction to Object-Based Reflective Concurrent Computation*, in *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, SIGPLAN Notices, 24(4)*, Agha G., Wegner P., et al., Eds., 1989.