

Contents

1	Introduction	1
1.1	Background Assumptions	1
2	Aspects	3
2.1	Basic Functionality	3
2.2	Optimizing Memory Usage	4
2.3	Cross-Cutting	7
2.4	Cross-cut Components	8
3	Aspect Oriented Programming	10
3.1	The Component Language & Program	11
3.2	The Aspect Language & Program	11
3.3	Weaving	12
3.4	Aspect Weaver	13
3.5	Related Work	14
4	Open Issues & Future Scope	15

Abstract

Aspect-oriented Programming (AOP), a paradigm invented at Xerox PARC in the 1990s, lets the developer better separate tasks that should not be inextricably tangled, such as mathematical operations and exception handling. The AOP approach has a number of benefits. First, it improves performance because the operations are more succinct. Second, it allows programmers to spend less time rewriting the same code. Overall, AOP enables better encapsulation of distinct procedures and promotes future interoperability.

There are 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. A general analysis of certain design decisions which are difficult to clearly capture in actual code, shows up as a regular pattern. These properties denoting decisions are called *aspects*, and show that the reason they have been hard to capture is that they cross-cut the system's basic functionality.

This basic idea was developed by *Gregor Kiczales* from the Xerox PARC to develop a brand 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.

Chapter 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 it has been found that there are 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.

But it is possible to develop programming techniques that make it possible to clearly express those programs that OOP (and POP) fail to support. The analysis of why some design decisions have been so difficult to cleanly capture in actual code, reveals a number of similarities. These 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 it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code. The basic concepts are beginning to take shape, and an expanding group of researchers are using them in their work. Furthermore, while AOP *qua* AOP is a new idea, there are existing systems that have AOP-like properties.

1.1 Background Assumptions

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 they will be referred to as generalized-procedure (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*. 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 in this case, other modes of decomposition are discussed.

Chapter 2

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 . 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.

2.1 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 follows:

```
(defun or! (a b)
  (let ((result (new-image)))
    (loop for i from 1 to width do
      (loop for j from 1 to height do
        (set-pixel result i j
          (or (get-pixel a i j)
              (get-pixel b i ))))))
    result))
```

Starting from the `or` and any other primitive filter, 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 minus 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)))

Table 2.1: Comparison

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.

2.2 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.

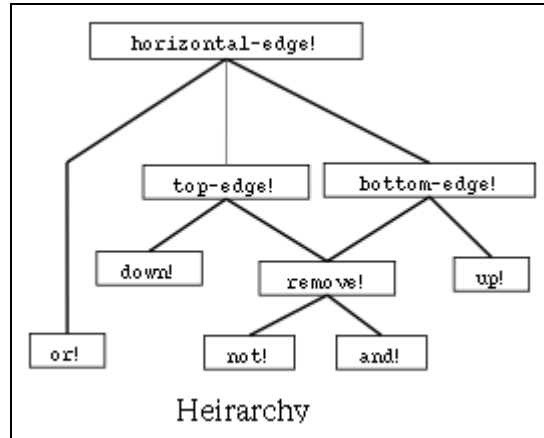


Figure 1

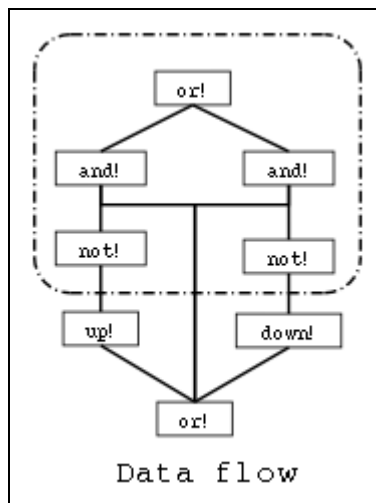


Figure 2

Two different diagrams of the un-optimized horizontal-edge! filter. The first 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.

2.3 Cross-Cutting

Returning to the example code, Figure 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—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.

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: 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.

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, 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.

2.4 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	minimizing network printers, traffic services synchronization constraints failure handling
matrix algorithms	procedural	linear algebra	matrix representation operations 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.

Chapter 3

Aspect Oriented Programming

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 . 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.

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,
- (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 aspect weaver for the combined languages,
- (iii.a) A component program, that implements the components using the component language
- (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).

3.1 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.

3.2 The Aspect Language & Program

The design of the aspect language used for this application is based on the observation that the dataflow graph 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 . 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.

3.3 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 #i... i refer to the edges coming into the node:

```
(pointwise (#<edge1> #<edge2>) (i1 i2) (or i1 i2))
(define-filter or! (a b)
  (pixelwise (a b) (aa bb) (or a b)))
(define-filter and! ... )
.....
.....
(cond ((and (eq (loop-shape node) ... )
  (eq (loop-shape input) ... )))
```

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*.

3.4 Aspect Weaver

Aspect weavers must process the component and aspect languages, 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. 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. 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. Thus the higher-level aspect language we have designed is implemented on top of a lower level one, as often happens in GP languages.

3.5 Related Work

- Aspect J Aspect Oriented Java
 - Aspect ++ AOP for VC++
 - Pythius Aspect Oriented Python
 - Trecc Aspect Oriented compiler tool
-

Chapter 4

Open Issues & Future Scope

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 it is important to begin work on this, given that it will take time to get solid results.

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 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 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 show, AOP can be used as an improvement to existing techniques. To fulfill this promise it must be developed in a way that integrates well with those techniques.
