ASPECT-ORIENTED PROGRAMMING

PRESENTED BY:

SASI VANDANA. T
II/IV
vandana_tamirisa@yahoo.com

RAM SANTOSH. K
II/IV
iamramsantosh@rocketmail.com

FROM THE DEPARTMENT OF INFORMATION TECHNOLOGY

KAKAKTIYA INSTITUTE OF TECHNOLOGY AND SCIENCE

WARANGAL
ASPECT – ORIENTED PROGRAMMING

ABSTRACT
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. 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 (AOP) 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 explaining the fundamentals and tools in aspect-oriented programming.

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 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. 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.

BACKGROUND ASSUMPTION
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. 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 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.

**WHAT ARE ASPECTS**

To better understand the origins of tangle 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. There are three implementations of the real application:

* easy to understand but inefficient,

* efficient but difficult to understand,

*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.

**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 pixel wise Logical or, might be implemented.

**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.

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. In short, revising the code to make more efficient use of memory has destroyed the original clean component structure.

In programs the complexity due to tangling quickly expands to become a major obstacle to ease of code development and maintenance. In real system this example was drawn from an important sub-component of an optical character recognition system. The clean implementation of the real system, similar to the first code, is only 768 lines of code; but the tangled implementation, which does the fusion optimization as well as memorization of intermediate results, compile-time memory allocation and specialized intermediate data structures, 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: 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 labelled a at the Bottom is the input image.

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!.

If we observe, the fusion horizontal edge! Oval does not incorporate all of 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 phenomenon is directly responsible for the tangling in the code. In 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.

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 cannot 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 he components in systemic ways. Examples of aspects include memory access Patterns and synchronization of concurrent objects. (Section 4 provides more examples of aspects.)

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!.
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.

**EXAMPLE OF HOW ASPECT 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.

<table>
<thead>
<tr>
<th>Application</th>
<th>GP language</th>
<th>Component Structure</th>
<th>Aspects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image processing</td>
<td>procedural</td>
<td>filters</td>
<td>loop fusion, result sharing, compile-time memory allocation</td>
</tr>
<tr>
<td>digital library</td>
<td>object-oriented</td>
<td>repositori es, printers, services</td>
<td>minimizing network traffic, synchronizat ion, constraints, failure handling</td>
</tr>
<tr>
<td>matrix algorithms</td>
<td>procedural</td>
<td>linear algebra operations</td>
<td>matrix representation, floating point error</td>
</tr>
</tbody>
</table>

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 fragments, because performance optimizations often exploit information about the execution context that spans components.

**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 PHASE 2. In PHASE 1 the weaver uses unfolding as a technique for generating a data flow graph from the component program. 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))

In PHASE 2 the aspect program is run, to edit the graph by collapsing nodes
together and adjusting their bodies accordingly.

**PHASE 1 A**

**PHASE 2 B**

**PHASE 3**

void main (int* i1)
{
...};
void loop1(int* i1)
{
...};

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.

**RESULTS**

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

```c
(cond ((and (eq (loop-shape node) ... ))
  (eq (loop-shape input) ... )))
```

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

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. 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 equations for this measure, as well as the numbers for this particular application are as follows:

\[
\text{Reduction in bloat due to tangling} = \frac{A - B}{C}
\]

here:

- A = Tangled code size.
- B = component program size.
- C = sum of aspects program size.

\[
= \frac{35213 - 756}{352}
\]

\[
= 98
\]

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.

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. 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 bandwidth 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. Failure handling using AOP is a future research GOAL.

THE COMPONENTS LANGUAGES & 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 preempt 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 preempt 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, \textit{Java} serves quite well as the component language. It provides an object model that implements the appropriate components, and avoids addressing

\begin{verbatim}
THE ASPECT LANGUAGE & PROGRAM
\end{verbatim}

The communication aspect. So, using Java as our component language 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 (footnote continued) gauge, the definition of two simple classes, books and repositories of books look like:

\begin{verbatim}
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 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)
  {
    …
  }
}
\end{verbatim}

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.

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.

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

**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 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 metaprogram, called at each method invocation, which uses the join point information and the aspect program, to know how to appropriately marshal the arguments. 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.

**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 programs 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.

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.

**SUBJECTIVE PROGRAMMING**
A natural question to ask is whether subjective programming 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 procedure. 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.

**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 methods let programmers draw different pictures of how objects should work.

**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 test bed 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.
REFERENCES:


