ASPECT ORIENTED PROGRAMMING

A CRITICAL ANALYSIS OF A NEW PROGRAMMING PARADIGM

T.J. Highley, Michael Lack, Perry Myers

Programming Languages – CS655 Semester Project

Professor Paul Reynolds

An aspect is not a something.
It is a something about a something.

TABLE OF CONTENTS

Introduction 2

1 – Aspect Oriented Programming Paradigm................................ 3

A Brief History Leading to AOP 3

Crosscutting Concerns, Aspects and Tangled Code................ 4

Aspect Oriented Programming and Weaving........................... 6

AspectJ Implementation 8

Dynamic Weaving and Aspect Instances...............................11

Explicit and Automatic Composition 12

Dynamic Weaving..13

AOP Design Philosophy...14

2 – AspectJ Critique...17

Comments on Dynamic Weaving..17

Comments on Protection Mechanisms..................................18

Comments on Abstract Methods...20

Comments on Exception Handling.......................................23

Comments on Aspects and Orthogonality..............................24

3 – AOP and Non-Object Oriented Languages 26

Do Aspects Apply to Procedural Languages? 26

Semantic and Syntactic Issues in Procedural Lang.................32

A Proposed Implementation for C...36

Discussion of Aspects in Procedural Languages.....................39

4 – Summary of Suggested Changes 40

5 – Concluding Remarks 41

Bibliography..42

Appendix – Correspondences with Xerox PARC.........................44

INTRODUCTION

The design and evolution of programming languages is one of the most important areas of computer science. Programming languages define the manner in which we communicate with our machines. They give us layers of abstraction with which to work, so we can accomplish our tasks without reaching into the hardware. They also attempt to increase our efficiency by automating many hardware bound tasks. Anyone who has attempted to write a large project in assembly language understands the necessity for higher level languages.

Over the last 50 or so years, languages have continued to evolve in order to support their ever-increasing usage. Program design and maintenance became issues with the dawn of software engineering. People sought to formalize methods for constructing correct, efficient and easily modified programs. Languages evolved in order to support these new requirements. Block structure grew into existence from a desire for modularity. Object-orientation (OO) was created from a desire to have language constructs for modeling real world objects and encouraging software reuse. OO has been around for more than a decade now, and has become the default paradigm in many peoples' minds.

Object-orientation still has its problems. While it encourages software reuse, practical experience has shown that OO does not handle this as effectively as people originally thought. Pre-packaged software often does not suit the programmer's needs, and ill-constructed interfaces make using these packages difficult. Furthermore, OO should enhance maintainability by causing redesign to affect as few modules/classes as possible. However, as our programs continue to get larger and larger it becomes increasingly difficult to cleanly separate concerns into modules.

Enter aspect-oriented programming (AOP). AOP could be the next step in the steady evolution of the OO paradigm, or perhaps it will evolve into a completely new paradigm independent of OO. Whatever the case, AOP offers a solution to a design and maintenance problem that has plagued software developers for years. That is, how can we create modules with little or no crosscutting concerns? AOP introduces the notion of Aspects, and shows how we can take crosscutting concerns out of modules and place them in a centralized place. While this paradigm is still relatively new, it seems promising and perhaps given time will replace OO.

This paper serves several purposes. The first section will attempt to give some background for AOP and describe some of the motivation in more detail. It will also briefly describe the syntax and semantics of AspectJ, a Java based implementation of AOP. We will also present a design philosophy for using aspects on top of an object-oriented paradigm. The second section discusses some problems we have observed in the AspectJ implementation. The third section discusses alternative paradigms to which AOP might apply. While current research seems to favor OO languages as the basis for AOP, this does not necessarily need to be the case. Procedural languages will be considered to see if they also might benefit from AOP. An implementation for C will be briefly outlined here. Perhaps in a later incarnation of this paper, this skeletal implementation will be more developed. Finally, we will give some concluding remarks about AOP and its usefulness.

1 – ASPECT ORIENTED PROGRAMMING

A BRIEF HISTORY OF DEVELOPMENTS LEADING TO ASPECT ORIENTED PROGRAMMING

Programming design principles began to emerge in the decade of the 70's as a result of the crisis created from increasing program complexity. This increased complexity, combined with a growing need for maintainable and evolvable programs led to the concept of *structured programming*. [Lop97] Structured programming is primarily concerned with finding, "a better organization of the program development process to achieve objectives such as simplicity, understandability, verifiability, modifiability, maintainability, etc." [Weg76] A related concept to structured programming is the idea of functional decomposition. This is the process of breaking down a problem into more manageable sub-problems during the design phase of programming. By creating these smaller functional modules and then recombining them in structured ways we achieve the goals of structured programming.

Structured programming was only the beginning of the software revolution. Program requirements continued to become more complex and involved more programmers. Issues like software reuse became prevalent, as people began to realize that they were potentially 'reinventing the wheel' each time they created a new program. Programmers had always reused their own software, but this was no longer sufficient as programs began to outlive their creators and span corporations. As Fred Brooks states, "The best way to attack the essence of building software is not to build it at all." [Bro95]

The idea of modularity seemed to be a good starting point for software reuse. Modularity, introduced along with structured programming, provided the programmer with clean interfaces to the functional units of a program. Even people other than the original programmer could reuse these units multiple times without modification (as long as the interface was well defined). However, a module's implementation was open and therefore subject to change at the programmer's discretion. This created even more problems in maintaining code, and the idea of a 'black box' was introduced. If the implementation could be *encapsulated* in the module and programmers could only see the interface, the implementation could remain constant. [Bro95]

However, what happens when a programmer does not want exactly what a module provides, but some variant of it that has a slightly different functionality? The programmer cannot modify the original module and he/she has no language constructs to easily reuse the module in another implementation. This need prompted the introduction of *inheritance*, which allowed programmers to extend or limit the functionality of a module without redefining the original module. These ideas were combined and evolved into our modern day conception of Object Orientation (OO).

OO seemingly solves many problems. It provides design and language constructs for supporting modularity, encapsulation and inheritance. It eliminates many of the above-discussed problems with software reuse and if used properly motivates the programmer to focus on good design before implementation. This could have been 'Silver Bullet' that Brooks discusses in several papers, but with the advent of OO we still have not seen the drastic increase in programmer productivity that was anticipated.

CROSSCUTTING CONCERNS, ASPECTS AND TANGLED CODE

One problem, according to Kiczales and other prominent researchers, lies in the existence of *crosscutting concerns*. That is, concerns which cannot be constrained easily into modular form. These concerns destroy the modularity that we strive for in our OO programs. They introduce related or even duplicated code into one or more modules. [Kic97]

Examples of crosscutting concerns are not hard to find in large systems. A group of crosscutting concerns that seem common to many programs already exists. Some examples of these are performance, synchronization, communication, graphics manipulation and debugging. Kiczales and several other researchers have published many papers illustrating examples of what constitutes a crosscutting concern. To read more about some examples of crosscutting concerns refer to [Kic97], [Lop97], [Men97] and [Aks98].

One example that was particularly useful was given in a presentation by Kiczales. A program called SpaceWar was constructed. It was written in Java with OO techniques and resembled the arcade game Asteroids. The user flew a spaceship, represented by a movable triangle form, and attempted to eliminate any enemy spaceships (same triangle form, different color) before they eliminated him/her. Each physical object in the field of view corresponded to a class definition (i.e. object) in the program. Some of the objects in the system were the user's spaceship, the enemy ships, bullets and energy packets. [AOP99]

In the original OO design, each object contained its own paint method. That is, the object encapsulated information relating to how it should be represented on the screen. Ships were represented as moving triangles, bullets as white dots and energy packets as larger blue dots. To change something specific about the way a ship or an energy packet behaved, one simply needed to modify the corresponding class. However, what if one wanted to change the entire 'look and feel' of the game? Perhaps the programmer wished to make the program more complex and use pre-drawn graphical images (JPEG's or bitmaps perhaps) for the object representations, instead of using the Java AWT primitives and stick figure designs. This change would involve modifying the paint method in every class that has one. [AOP99]

Each class encapsulates a functional component of the system. There are other concerns that are not addressed by individual modules, but rather by crosscutting sections throughout multiple

modules. The programmer can address this concern in one of two ways. Either he/she can implement a paint method in each of the affected classes, or he/she can create an "intermediate" class that intercepts an object's call to paint and executes the correct method based upon the calling object's type. The first method is efficient, but it results in a need to touch each class should the look and feel need to be changed. This becomes a nightmare in software maintenance. The second option encapsulates the look and feel into a class and therefore has a well-defined interface and locality, but is inefficient. Each time an object wishes to paint itself on the screen, a method call to another object must be made. [Kic97] Another problems is that there is no way of restricting the set of objects that can access the member functions. Perhaps there are several definitions of paint, one for each type of object that needs to be displayed. What prevents a ship from calling paint for an energy packet? This lack of access control to member functions destroys the interfaces that are provided in OO.

It should be clear from both the example above and in the papers cited, that crosscutting concerns do exist in programs, and their representation can cause problems with both maintainability and efficiency. In some situations, redesign of the system might transform a crosscutting concern into an object. That is, the crosscutting concern is now no longer crosscutting; it is cleanly encapsulated in a module. One might be tempted to think that this method of module re-factoring might eliminate this problem altogether. Kiczales argues (but does not prove) that there are invariably some situations where no matter how you re-factor the system some crosscutting concerns will exist. Although a formal proof to show this does not exist, it seems intuitive that this is true. Therefore, the remainder of this paper will be written under the assumption that crosscutting concerns exist and cannot be simply re-factored out of the design in all situations.

An aspect then is a representation of a crosscutting concern. A component is a modular unit of functional decomposition, which addresses a specific concern or function of the system. An aspect is similar, in that it addresses a concern of the system, but it cannot be cleanly decomposed into a component. Kiczales provides some more concrete definitions to differentiate between aspects and components. He states that a property that must be implemented is:

> **A component, if it can be cleanly encapsulated in a generalized procedure.** By cleanly, we mean well localized, and easily accessed and composed as necessary.
>
> **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.
>
> [Kic97]

Another definition of aspect is provided by Czarnecki. A *domain* is defined as an area of expertise or more formally, "an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area." [Cza98] Program design involves capturing the domain and presenting it in an acceptable form. For example, OO models operational domains by capturing functionality in classes. *Decomposition* is the process of breaking down a problem into a set of smaller problems. Problem solving is simply a process of capturing the appropriate domains, by breaking down the problem into smaller problems and then recomposing the solutions in a way that satisfies the original specifications. A domain that defines the manner in which the original problem should be decomposed is called a *concern*. Using this terminology, an *aspect* is a partial representation of some concepts that relate to a concern. [Cza98] This method of defining the meaning of an aspect results in a definition that seems similar to the one above, but is defined purely in terms of Domain Engineering. This just provides an alternate perspective from which to look at aspects.

Crosscutting concerns wreak havoc on the maintainability of code if they are not handled properly. If one of these concerns needs to be changed at some point in the program's lifetime, multiple (sometimes all) modules need to be modified (as in the SpaceWar example). Design by functional decomposition often results in programs that do not adequately satisfy one or more aspects. These programs are re-worked by hand to satisfy the aspects, and in doing so the code often becomes cluttered. Just as indiscriminate usage of the goto caused "spaghetti code", as Knuth described it, the re-working of the code to satisfy aspects creates "tangled code" in Kiczales and Lopes' terminology. [Lop97]

Lopes' explanation of why aspects exist is very concise and summarizes the problem nicely. She states that functional components (i.e. modules) can be composed together via the 'uses' relation. Modern software systems are complex enough that, "many issues that must be programmed relate to other parts of the system in sophisticated ways that the 'uses' relation doesn't quite capture." [Lop97] The existing composition mechanisms provided in programming languages do not adequately represent these issues in the program.

The programmer has a choice between two implementations. She can choose the one that is clear, easy to modify and represents the design well. Conversely, she could choose the one that better satisfies one or more aspects of the program. With the second type of implementation, tangled code is the result. This is because it attempts to address crosscutting concerns without support for representing these concerns in a modular way. Most languages provide mechanisms for composing units of functionality together but do not provide support for co-composing functional units with aspects. It is the process of manually co-composing which results in the tangled code. [Lop97]

ASPECT ORIENTED PROGRAMMING AND WEAVING

Now that the nature of the problem is understood, the next logical step is to provide a solution. Kiczales' solution is to provide support in the language (or provide another language) for defining aspects along with the already present support for defining components. This new approach to

programming is known as Aspect Oriented Programming (AOP) and is still in its infancy. AOP puts a greater focus on crosscutting concerns than is present in OO or many other language paradigms. It allows aspects to be cleanly separated and placed into modules that can be composed with other components (including other aspects) in the system.

AOP began as an experimental language framework, called D, created as part of Lopes' thesis work. The language D contained three components, a language for defining components (like Java or C) and two special languages for describing aspects of the problem domain: COOL and RIDL. Lopes was specifically working with a problem domain that had synchronization and communication as crosscutting concerns. These crosscutting concerns were modeled as aspects through the two aspect languages. COOL modeled thread coordination and RIDL modeled remote access. The component definition language was Jcore, which was a subset of Java. [LoK97]

The two aspects were written in their corresponding aspect languages and the other components were written in Jcore. A preprocessor then combined the three programs and produced a complete Java program through a process called *weaving*. [LoK97] The weaving process is similar to the hand optimization process that creates tangled code. The aspect languages and the weaver simply allow the programmer to specify *how* the optimization should be done, and where code should be changed. However, the programmer need not be concerned with the woven code, just as he/she is not concerned with the intermediate results of a compiler. The woven code is never modified. Modifications are made in the original programs (Jcore, RIDL and COOL) and then the programs are re-woven and re-compiled to create an executable program.

Another piece of terminology related to weaving is the *join point*. A join point is a location that is affected by a crosscutting concern. [Oss98] The place where a weaver inserts aspect code is a join point. Join points can be present at either the *statement level* or the *operation level*. Statement level implies that the set of possible join points include every statement (line of code) in the system. Operation level implies that the possible set of join points includes every operation (method invocation) that the system performs. [Oss98] As will be seen later in this section, the current implementation of AOP uses operation join points instead of statement join points.

It would seem from the example that the concept of aspects and crosscutting are tied to OO. This is not the case. While it appears that AOP has grown out of the OO paradigm, it is more pervasive than that. An aspect is simply a concern in the design of a system that cannot be cleanly captured into a unit of modularity. This unit could be a class (as in OO), a procedure (in the procedural paradigm) or a function (in the functional paradigm). The important idea to remember is that the mechanism for breaking the original problem into sub-problems resulted in modules that share a common concern.

In [Kic97] and [Men97] an image processing system is described and programmed in Common Lisp. An aspect of this system is discovered (performance through loop-fusion) and is programmed in a designed aspect language. The component and aspect programs are 'woven' together and a C program is output. [Kic97] Notice that in this example there is no mention of objects or object orientation. Although Kiczales is eager to point out that aspects are not constrained to OO, not much information about support for aspects in procedural or other types of languages is given. Section 3, "AOP and Non-Object Oriented Languages" will go into more depth on this issue.

The idea of creating a separate language for each aspect eventually was replaced by the idea that support for defining generic aspects should be included in the language or added in as an extension to the language. A generic aspect weaver could then be implemented and reused regardless of the type of aspect designed. This is in contrast to having to create a separate aspect weaver for each of the custom aspect languages (like COOL and RIDL). This resulted in the eventual subsuming of other aspect definition languages into AspectJ, a generic aspect programming language built on top of Java.

ASPECTJ IMPLEMENTATION

AspectJ is one implementation of AOP. AOP is not limited to AspectJ in the same way that OO is not limited to Java or C++. It is the first attempt at a general AOP language, and is built on top of Java. This section will attempt to describe some of the syntax and semantics of AspectJ in more detail so that the reader will have a solid foundation with which to understand our arguments in Section 2.

In AspectJ the definition of an aspect is very similar (in some cases identical) to the definition for class. A class contains data and operations (variables and methods, respectively). Classes can be instantiated to create Objects. Classes can be defined to inherit from other classes. Classes have multiple protection mechanisms at their disposal in order to protect/hide their representations. Aspects have all of these features and add one more. They have the ability to enhance the behavior of other classes through a mechanism called weaving. To summarize, classes contain variables and methods, while aspects contain variables, methods and weaves. [AsJ99]

The syntax for declaring an aspect is very similar to the syntax for declaring a class. The basic syntax looks like the following:

```
aspect <aspect_name> {
   // The following two components have the same structure as a normal
   // Java class declaration
   <variable declarations>...
   <method declarations>...

   // The following lines are specific to aspects

   advise <designator> {
      [static] before {
         // Code to be executed before the method begins goes here
      }

      [static] after {
         // Code to be executed after the method ends goes here
      }

      [static] catch (<exception type> e) { <exception handler> }

      [static] finally {
         // This is executed after the method is finished executing
         // regardless of whether or not an exception was thrown
   }

   introduce <class>.<variable declaration>; ...

   introduce <designator> {
      // the body of the method
   }
} // end <aspect_name>
```

Designators are a way of specifying the methods or classes that are affected by a weave. A designator can be either a method, a constructor or a field. The designators allowed for introduce weaves are very simple. They can be methods, constructors or fields where all information is specified. Therefore, an introduce statement can have the following forms:

```
introduce <type> <class>.<variable> = <initialization>;
introduce <modifiers> <return type> <class>.<method>(<parameters>) {}
introduce <class>(<parameters>) {}
```

An introduce clause can contain multiple designators as long as they are of the same type. Constructor designators cannot be mixed with method designators. This allows the same method/constructor to be used by multiple classes. Exactly *how* the classes use the method from an introduce will be discussed later.

In the case of advising, advice can be specified to be run before the method executes, after a method's execution or both. If the before or after advice throws an exception, an optional catch can be used. Advice can also be specified as **finally** advice, which is similar to the finally keyword in Java: it will run even if the exit from the method is due to an exception. Advise has the same restriction on combining designators as introduce. Multiple designators can be used as long as they are either all methods or all constructors. The optional keyword **static** can be applied to the before and/or after statements. Static weaves are relatively simple to describe, but if the keyword static is

omitted the result is dynamic weaving which is much more complicated. The difference between static and dynamic weaving will be discussed shortly.

Additionally, wildcards (*) can be used in place of explicit names in all places except the class designator. The wildcard for the method modifier is simply omitting it entirely. For example:

```
advise public & !static * Foo.*(*)
// Advise all public and non-static methods in Foo.

advise int Foo.bar(char fooBar)
// Advise all methods named bar in class Foo that return an int and
// take a char as a parameter.
```

The syntax for weaves is quite simple; it does not seem to be terribly difficult to define the aspects once you have identified what the aspects of a given system are. The bulk of the work is in the design process and identifying what the aspects are and which classes aspects should affect.

One thing to keep in mind when programming aspects is that the methods that are introduced and the sections of methods that are advised share scope with the aspect and their associated class. Therefore, if an aspect Foo introduces a method A into class Bar, A has scope in both Foo and Bar. That is, it can access all of the other methods and data members.

Advising and introducing into interfaces is also slightly different than working with classes. It is possible to introduce methods, fields and constructors into an interface. However, because an interface is devoid of implementation and data members, the only thing introduced into an interface is the method's signature. The implementation is actually introduced into all classes that implement that interface. If a field was introduced, it is simply introduced into all classes implementing the interface. This feature is powerful, as it allows for multiple inheritance in Java. [AsJ99] Nevertheless, it also seems dangerous because multiple inheritance was considered and not included in Java for many reasons, and reintroducing it seems both counterintuitive and a violation of the principle of security [Mac87].

As mentioned before, advised and introduced methods have scope in multiple places. The order of scope resolution in AspectJ is the following:

1. Local variables and classes declared in the weave
2. Parameters of the advised methods, as named in the formal parameters of the weave
3. Variables, methods and classes defined locally in the aspect
4. Variables, methods and classes defined locally in the class [AsJ99]

AspectJ includes several variables to be used in case of name shadowing. These are thisJoinPoint.aspect and thisJoinPoint.object. These variables represent the given aspect instance in question and the object that is currently invoking the advice or introduced method in the aspect. The name of the current method whose advice is being executed in is referenced by thisJoinPoint.methodName. The name of the current class that is executing the advice is referenced by thisJoinPoint.className. Finally, the return value of a given method is stored in thisJoinPoint.result. [AsJ99]

In the case of static weaving (i.e. advice with the keyword static used) the advice is executed for every instance of the class and every invocation of the method. In contrast, dynamic weaving allows advice to be applied to only certain instances of the class and advice can change throughout the course of an object's lifetime. Because dynamic weaving is such a complex topic, and much of our critique of AspectJ concerns it, the next few sections will be devoted to explaining it.

DYNAMIC WEAVING AND ASPECT INSTANCES

What is an aspect instance? It is easiest to first examine what an instance is. Let us consider the more familiar concept of a class and class instance. A class instance, of course, is better known as an object. What does it mean to say an object is an instance of a class? The class defines the form of the object, but the object is a distinct entity of its own. There may be several objects of the same class. That means that they have the same form but they are each distinct entities. Each distinct object of the same class has its own values for the variables defined in the class, but each distinct object has the same methods available to be called on it.

An aspect instance is similar to an object in many respects. One can be created just like an object is created:

```
Aspect foo = new Aspect();
```

The above line of code creates a new aspect instance and binds it to the variable foo. Aspect instances and objects are so similar that if the aspect has no weaving built into it, AspectJ treats an aspect instance and an object as equivalent entities. (In this case, one might wonder why an aspect was used in the first place.) In AspectJ, an aspect is simply a class with crosscutting capabilities; those capabilities are found in the weaves. If an aspect does not utilize those crosscutting capabilities, it really is just like a class. Considering an aspect merely a special type of class may or may not be a wise decision, though. This will be investigated further in a later section.

An aspect instance, then, has its form determined by an aspect but is a distinct entity of its own. Analogous to classes and objects, each distinct instance of the same aspect has its own values for the variables defined in the aspect, but has the same methods available to be called on it. Beyond that, each aspect instance may have its own particular set of join points associated with it.

When discussing aspect instances, there is an innate tendency to abbreviate. An abbreviation from "aspect instance" to simply "aspect" is unacceptable because "aspect" denotes the module that determines the form of an aspect instance, not the aspect instance itself. On the other hand, an abbreviation to "instance" is also unacceptable because it is ambiguous. There can be instances of many things, not just aspects. (Class instances are just one example.) For this reason, we suggest that aspect instances be referred to as **aspins**. This creates a simple one-word notation to represent aspect instances, just as the word object represents class instances. We hope that this terminology is adopted by the AOP community, and we will be using it throughout the rest of this paper.

Let us take a closer look at aspins in AspectJ. An aspect without weaves is just like a class; therefore, the weaves are obviously the distinguishing features of the aspect. It is not always necessary to instantiate an aspect for it to be useful because the weaves can be used to statically affect classes. This is different from a class because a class that is not referenced anywhere else in the program is simply unused. The aspect, merely by being compiled with a class, can change the class by giving it advice or introducing new variables or methods. Any objects instantiated from the affected class will include the changes imparted by the weave. This only applies to introduce weaves and static advice to methods. The other alternative is advice that involves dynamic weaving and aspins.

EXPLICIT AND AUTOMATIC COMPOSITION

Before discussing dynamic weaving, it should be noted that the use of aspins does not necessarily mean that dynamic weaving will be used. There is a manner of using aspins without dynamic weaving. The AspectJ primer refers to this as explicit composition. With explicit composition, a variable of the aspect's type can be placed in a class through an introduce weave. Whenever the class is instantiated, the resulting object will have an aspin as one of its data members. Through static advice to the class, the aspin can be appropriately modified whenever there are calls to methods of the object. The primer makes a note that it is an important style issue to not include any "gluing" code like the introduced aspect variable in the actual class. All of the "gluing" code should be in the aspect itself. [AsJ99] We argue even more strongly for this issue later.

That, then, is explicit composition. Another method of using aspins involves automatic composition and dynamic weaves. Automatic composition refers to when the binding takes place between an object and an aspin. In explicit composition, the binding takes place at compile time, because it is explicitly stated that any object of class A will have an aspin of aspect B. In automatic composition, the binding takes place at run time. The following example from the AspectJ primer illustrates automatic composition. Binding does not occur until the run-time invocation of the addObject() method. In AspectJ terminology, the addObject() method places the parameter object into the domain of the aspin. [AsJ99]

```
aspect PointShadow {
    int x, y, offset;
    PointShadow(int _x, int _y, int _offset) {
        x = _x + _offset;
        y = _y + _offset;
        offset = _offset;
    }

    advise Point(int _x, int _y) {
         static after {
              PointShadow psh = new PointShadow(_x, _y,
                                  10).addObject(thisJoinPoint.object);}
    }

    advise void Point.setX(int _x) {
        after { thisJoinPoint.aspect.x = thisJoinPoint.object.x +
                                         thisJoinPoint.aspect.offset; }
    }
    advise void Point.setY(int _y) {
        after { thisJoinPoint.aspect.y = thisJoinPoint.object.y +
                                         thisJoinPoint.aspect.offset; }
    }
}
```

DYNAMIC WEAVING

Advice that is not static does not affect a class itself. Rather, it affects only specific instances (objects) of the advised class. Also, it is not the aspect itself that is giving advice to the specific objects, but certain instances of the aspect (aspins) giving advice to certain objects. AspectJ provides the aforementioned addObject() method, which is available in aspins. It places an object in the domain of an aspin. The non-static advice from an aspin will be executed whenever the associated method is invoked on an object in its domain. If the same object is in the domain of more than one aspect instance (which is quite possible), then whenever a method is invoked on that object, all of the relevant pieces of advice from the various aspins will be executed. There is an obvious question here, though. In what order will the pieces of advice be executed? This topic is slightly out of the scope of this paper, and will not be discussed further.

When AspectJ weaves an aspect, all of the advice is actually weaved into the advised class. This includes the non-static advice. In that sense, the term dynamic weaving is a bit of a misnomer. The weaving itself is done statically at compile time, but the decision of whether or not to execute the weave, and how many times to execute the weave, is done dynamically at run time. The weaves are all moved to their appropriate classes while the aspect itself is transformed into a class which has any aspect-specific methods or variables as well as a small set of special methods: getObjects(), addObject() and removeObject(). After the weave, the aspect-turned-class also has a very important data member that is a vector of objects: the list of all objects in the aspects's domain. This list can be retrieved via the getObjects() method and modified by the methods addObject() and removeObject().

The advised classes also have a new data member added to them, in addition to the advice modifications. This data member, _aspects, is a vector which contains all of the aspins that have the object in their domains. This list of aspins is also modified by calls to addObject() or removeObject(), and can be retrieved through the getAspects() method on an advised object.

These data structures are used to implement the non-static weaves. The actual execution of the non-static weaves is carried out via the following mechanism. Whenever a method is invoked, all static advice is run, as previously discussed. The code for the non-static weaves is also present, but access to it is guarded by checking the _aspects vector. Each aspin in the _aspects vector is checked. If an aspin's aspect type matches the type associated with a weave, that piece of advice is run on that object for that aspin. For more information see the PointShadow example in [AsJ99]. The example illustrates that dynamic weaving is particularly useful when several (possibly different types) objects need to be associated with one particular aspin. [AsJ99]

AN ASPECT ORIENTED PROGRAMMING DESIGN PHILOSOPHY

While examining aspect-oriented programming, and especially AspectJ, it became apparent that AOP ignores many of the principles that are central to OOP. Among these are separation of concerns and information hiding. Examples of these principles were given in the previous section discussing AOP and its emergence into the programming language community.

AOP introduces a new type of module: the aspect. This new type of module, however, is very different from an object. It has the ability to break all the rules of object separation. We propose a philosophy of aspects that allows an aspect to crosscut the object hierarchy while still maintaining a separation of modules. There are four points to this philosophy:

1. An object is a something.
2. An aspect is not a something. It is a something about a something.
3. Objects are not dependent on aspects.
4. Aspects represent some feature or property of objects, but they do not have control over objects.

We hold these truths to be self-evident but we will defend them anyway.

1. An object is a something.

   An object exists on its own. It is a something. How does one determine what a particular object is? A programmer should be able to look at the code for a class and determine what the object is (the variables that make up the object) and what the object can do (the methods of the object). Some of what it is and what it can do may be based on a superclass, but that is also evident from the definition of the class. It is based only on the interface of the superclass, not on its implementation. An object may also be associated with a number of aspects. Those relationships are determined by the aspects, not the object. If the object was not associated with those aspects, it would be a less detailed object, but it would still be an object. An object is an entity in itself.

2. An aspect is not a something. It is a something about a something.

   An aspect is written to cleanly modularize a crosscutting concern. This concern, by definition, cuts across a number of different components. In object-oriented programming these components are called objects. If an aspect is not associated with any class, then its concern cuts across zero classes, and thus the aspect is meaningless in this context. Therefore, it does not make sense to talk about utilizing an aspect without keeping in mind the classes that it crosscuts. It does make sense to discuss what an aspect is capable of providing, but the aspect does not have functionality unless it is actually applied to a class. They are not functional units themselves and should not be treated as such.

3. Objects are not dependent on aspects.

   An aspect should not change the interfaces of the classes it touches. It should only augment the implementations of those interfaces. Because it only affects the implementations of the classes, and does not change the interfaces of the classes, encapsulation is maintained. The classes retain their original black box interfaces, though the insides of the boxes may be changed. The inclusion of an aspect in a program should (and does) affect the behavior of the objects in the program, but it should not be required to enable the objects to be functional units. The classes of a program should be functional units by nature of their design. "Components tend to be units of the system's functional decomposition." [Kic97]

4. Aspects represent some feature or property of objects, but they do not have control over objects

   Aspects are over and above objects. That means that they can violate information hiding in certain ways because they can know things about an object that are hidden from other objects. However, they should not intrude upon the internal representation of an object any more than other objects are allowed to intrude. Aspects should be allowed to have this special view of many objects, but should be bound to manipulating objects in the same way that other objects do, through their available member functions.

This philosophy of design grew out of an attempt to balance a number of generally accepted design principles that tend to work against each other. In particular we want to enable aspects to do their job of providing abstraction and automation, while still maintaining the principles of information hiding and manifest interface. [Mac87]

Let us for a second step out of the realm of computer science and into a different world...

In this world, there are deaf hunchbacks (objects) living in houses. From these houses, messages are occasionally sent to other houses. Each house may have a different number of doors, but messages are only accepted at a mailbox by a door. Being hunched over, the people never look up. Being deaf, they do not hear. They are often unaware that there are dragons (aspects) flying overhead. They don't realize that some of the houses they send messages to are owned by those dragons. The hunchbacks are also unaware that the ceilings of their own houses are made of glass. The flying dragons see the hunchbacks placing messages in various mailboxes. Occasionally, a hovering dragon may fly down and take a piece of mail out of a mailbox, unbeknownst to the hunchbacks. While deciding what to do with the stolen piece of mail, the dragon may peer down into a house through the glass ceiling. She is not able to swoop down into the house, but she enjoys her special view. The dragons may choose to do a number of different things. They may go paint a house, for example. The dragons also have a code of honor. They may read someone else's mail, but they would not steal it. They always eventually put it back in the proper mailbox. Every once in a while, the dragons go to a neighboring state for the National Dragon Festival (NDF, because the wizards who created the dragons have to use acronyms in order to survive). While the dragons are gone, the hunchbacks continue to go about their business as usual. They notice that the houses stop changing colors (since the dragons are not painting), but this does not affect their lives at all. When the dragons are around, they do their best to keep the hunchbacks from bumping into each other. While the dragons are at the NDF, the hunchbacks fall down more. Despite the lack of color and all the falling down, the hunchbacks' messages are still delivered and received just like always.

Our fanciful land of hunchbacks and dragons illustrates our philosophy of the relationship between objects and aspects. The dragons are still trying to understand how best to relate to the hunchbacks; the hunchbacks are not offering much insight.

They will all live happily ever after anyway.

2 – AspectJ Critique

The philosophy we have presented in the previous section gives a rough framework that attempts to keep aspect-oriented programming from violating the structure that object-oriented programming provides, while still allowing aspect-oriented programming to accomplish its goals. That is, to provide support for co-composing crosscutting concerns with components, while maintaining the integrity of the components themselves. The current implementation of AspectJ violates some of our stated principles. Many of these violations result from intentional decisions to give freedom to the developing user base and allow "them more room to experiment and discover what really is right." [EMa99] We offer some suggestions, which we attempt to base more on the accepted principles of programming language design rather than upon personal opinions of the usefulness of language constructs. We examine places where AspectJ violates these principles and offer some ideas on how these violations might be resolved.

COMMENTS ON DYNAMIC WEAVING

Dynamic weaves are a powerful mechanism, but the capabilities they provide seem like self-modifying code. In AspectJ, a method call involves more than simply the method body. It also involves the advice associated with that method call. If a program can change its own advice patterns and then execute a method with advice that it had tailored for itself, then it would seem that the program has succeeded in modifying and then executing its own code. In reality, it is quite a bit of a stretch to say that AspectJ supports self-modifying code. Specifically, AspectJ does not quite fit the description as outlined in [Mod92]. The dynamic modifications do not actually involve modifying lines of code; they only involve dynamic decisions of where certain specific sections of code are located. Even this capability can be dangerous, though. We will illustrate with an example: the third-party modifier.

Consider an object that instantiates an aspect and adds itself to the aspin's domain. The instantiating object may then invoke a method on a different object. This "third-party" object may use the getAspects() method to access the aspin that was created by the first object. Once it has access to the aspin, it can use addObject() to put any object it wants into the aspin's domain. This may not seem like a major problem, but consider a large system. If there is an instantiation of an apect, it is reasonable to ask the question, "What objects may potentially be inserted into the domain of this aspin?" Where should a programmer have to look to be sure he found all of the relevant addObject() calls? If there is no restriction on where addObject() may be called from, the programmer actually has to check the entire system, which is unreasonable in a very large programming environment. The abuse of addObject() and removeObject() is what makes dynamic weaving seem like self-modifying code. Failure to appropriately control their usage can lead to the potential nightmare described here.

It makes more sense (and is more feasible in an extremely large system) if modifications to the aspin's domain are limited to the code in the aspect itself. The example described above demonstrates one motivation for restricting access to the addObject() method to the aspins. Another motivation is philosophical point number 3: the object should not be dependent on the aspect. If an object is making a call to addObject(), it is utilizing capabilities that can only be used in the presence of aspects. The simple instantiation of an aspect in an object actually makes the object dependent on the aspect. For that reason, we suggest that aspect instantiation also be restricted to code that originates in the aspect.

COMMENTS ON PROTECTION MECHANISMS

To what extent are aspects part of a class? An aspect cuts across classes. The part that cuts across a class can be considered part of the class itself. After all, in the AspectJ implementation, the weaver actually makes the woven sections part of the classes they cut across. If part of an aspect is really part of a class then maybe it should have full access to the class data members. On the other hand, given that the aspect and the class are different entities, allowing the aspect full reign over the class' data members seems to violate information hiding. It also violates our fourth philosophical principle: aspects represent some feature or property of objects, but they do not control the objects. If an aspect has full reign over an object's data members, then it can control the object, and the functionality of the object can potentially be dependent on the aspect.

It is easy to argue that an aspect needs to be "over and above" the objects. Consider a synchronization aspect, which monitors access to several guarded data structures. This synchronization aspect needs to know which methods access these data structures, even if those methods are private methods. In order to see those private methods, the aspect must be "over and above" the normal protection schemes of object-oriented programming. Without full knowledge of all of the methods that access the protected data structure, there could be a conflict if two unguarded accesses are attempted at the same time. This is an argument for allowing the aspect to have knowledge of private information, but not an argument for allowing the aspect to control the private information (i.e. modify the information). For example, the synchronization aspect delays the execution of a private method until the appropriate data structures are available, but the aspect does not need to modify the method itself.

We propose that aspects be permitted to view class private elements, but not be permitted to change them. Whenever it seems that an aspect needs to modify a private element of a class, several questions should be asked. Does it make more sense for this private data member to be public? If an aspect needs to reach in and change the private element, is it also reasonable to expect object to have a valid reason to reach in and change the element? If so, the element should be made available through the addition of inspectors and mutators. Does it instead make more sense for this private data member or method to be migrated to an aspect? In an aspect, it may simply be a data member of that aspect or it may be introduced by the aspect into the class. If no other object would ever have reason to modify the private member, it may make sense for that member to be part of the aspect instead of the class. Perhaps the aspect should be refactored into a

class and related to the other classes through some inheritance mechanism. However, the following example illustrates a remaining difficult case.

There are several classes, and each class has a method that is related to a method in the other classes. We would like to collect these methods into an aspect to capture the crosscutting concern. The methods all change data members which are private to their respective classes. We want the methods and the private data to be located together so that the state changes of the private data can be seen easily. (See philosophical point number four.) However, the private data members are also modified by other methods within their respective classes, so moving the data members to the aspect is not a viable alternative. We now have three design goals competing against each other: First, we want to capture a crosscutting concern into one place. Second, we want classes to be independent of the mechanism that captures the crosscutting concern. Third, we want classes to remain encapsulated entities, where we do not reveal to other objects any more than is necessary and private data is not modified by the aspect.

We propose two possible solutions to this problem. A first solution is delegation with access-control lists. With this solution, each class continues to have its respective method, but the method's body becomes a stub that merely invokes a method in the aspect (where the actual work is done). The stub passes as parameters to the method in the aspect any private data members that need to be modified. Because the stub is present and its body is merely a method invocation, two important properties are provided. First, it is still possible to look at only the class and tell when the private data members are modified because the private data members are passed as parameters. Aspects do not need to be able to reach in and touch the private data. Second, because the body is merely a method invocation, the class is not dependent on any knowledge of the aspects. It is dependent on the method, but in the absence of aspects the method could be provided by a class. The bodies of all the methods are collected into the aspect, thus capturing the crosscutting concern. The only question remaining is whether we have revealed more than is necessary. If the methods were originally private, we have. All of the methods are now collected into a single aspect, but the methods in the aspect are visible to everyone. (Note that these are just methods in the aspect, not methods that the aspect introduces somewhere else.) If these methods were private in the first place, we should not make them visible to everyone now. Here, we propose the inclusion of access lists similar to what will be described in detail in section three. The access lists will ensure that the only class that invokes the method is the class that originally contained it.

There is still one objection to this solution, which was raised earlier in this paper. What effect does this solution have on efficiency? This solution requires an extra method invocation. This is the main reason why the 'intercepting' class was rejected as a solution for crosscutting concerns. We offer this solution recognizing that it is a variation of the already rejected "intercepting" class. We would like to mention two things about this solution and its efficiency. First, the modularity and independence of code that this method provides is more important than the lost efficiency of an extra method invocation. Second, compilers could be optimized to recognize this delegation for what it is and actually substitute the body into the stub to remove the extra method invocation. Kiczales argues in [Kic97] that an advantage of AOP is that it does not rely on smart compilers, and that the "weavers' job is integration, rather than inspiration." However, this compiler optimization seems simple in comparison to the optimizations he was considering.

A second solution is to allow aspects to modify both public and protected members of classes. This gives the aspect a relationship to classes that is similar to subclasses and other classes within the same package. A flaw in this solution is that there may be circumstances where an aspect should have access to a member that the subclasses or other package members should not have (or vice versa). If applicable, it may be possible to make the class **final** to prevent subclasses from accessing protected members, since there would be no subclasses. The problem with restricting access to packages when access by an aspect is needed is a difficult one. Either the delegation method discussed above could be used, or perhaps another level of protection could be specified in the Java language. While adding more syntax could make the language overly complex, it would clear up a lot of this confusion. We feel intuitively that the need for this specific protection scheme would not happen very often. In any case, we hold that an aspect should not be permitted to modify either the values of private data members or the functionality of private methods.

COMMENTS ON ABSTRACT METHODS

In the current implementation of AspectJ, an abstract method that is defined in a superclass may be supplied either by a subclass or by an aspect that advises a subclass. If an abstract method is supplied by an aspect, then the class is incomplete without that method and the class is then dependent on the aspect. We hold that the class should not be dependent on the aspect, and therefore suggest that the subclass should always implement the abstract method if it is part of the signature of any ancestor.

It is very likely that the different implementations of an abstract method will all be related. In that case, it makes sense to have those implementations collected into an aspect. However, if all of the implementations are in the aspect, it also makes sense to have the declaration of the abstract method itself introduced from the aspect. By doing this introduction, the removal of the aspect

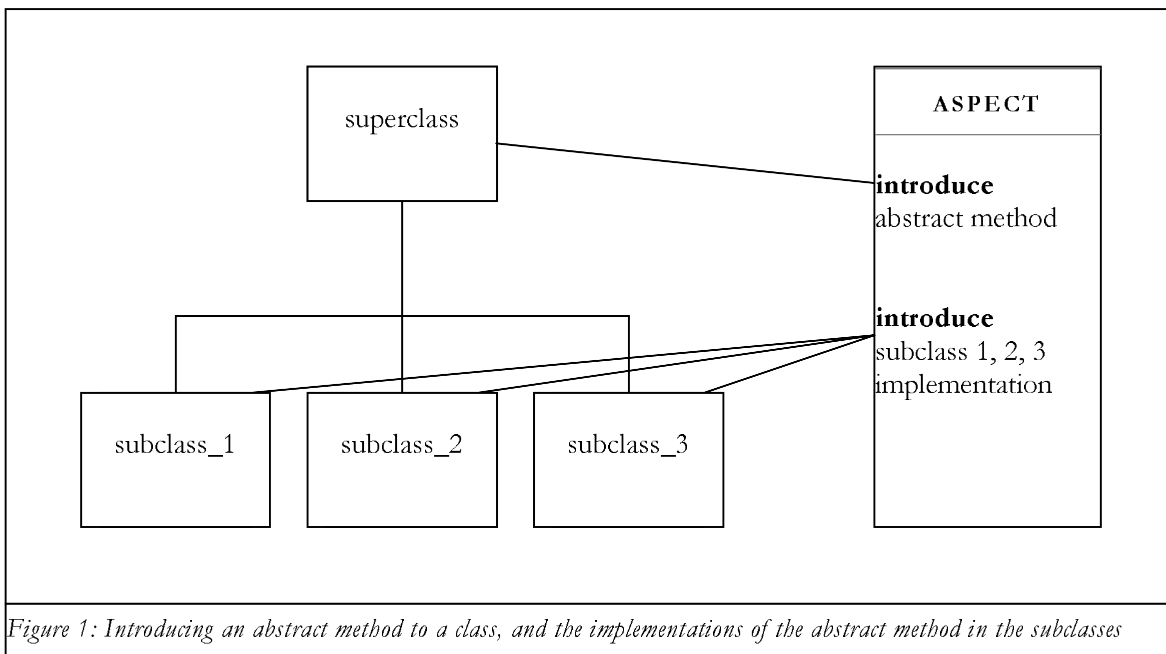

Figure 1: Introducing an abstract method to a class, and the implementations of the abstract method in the subclasses

would still remove the implementations of the abstract method, but it would also remove the specification that the method was necessary. The class, therefore, is not dependent on the aspect in this design.

It is less likely, but still possible, that some implementations of an abstract method will be related while others are not related. The programmer may want to group a subset of the implementations of an abstract method into an aspect, while excluding other implementations. Our proposition above will not work. In this case, the superclass can declare the abstract method. Some subclasses will contain an implementation for the abstract class. Some implementations will be in an aspect. However, the implementation is not provided through an introduce weave. Instead, we propose a new type of weave: an instead weave. In order that the subclasses are still complete, they should all have an "implementation" for the abstract methods of the superclass. Some of those implementations, however, should be null implementations. An instead weave will replace a null implementation with the implementation described in the aspect. Any attempt to give instead advice to a method that has a non-empty body should result in an error.

An alternative would be to use the introduce syntax but have the weaver automatically replace the null implementation rather than giving a multiply-defined error. The instead syntax is more explicit, but the introduce syntax would keep the language syntax to a minimum. In any case, all abstract methods declared in a superclass should (for completeness) be mentioned in the subclass, and not merely introduced by an aspect.

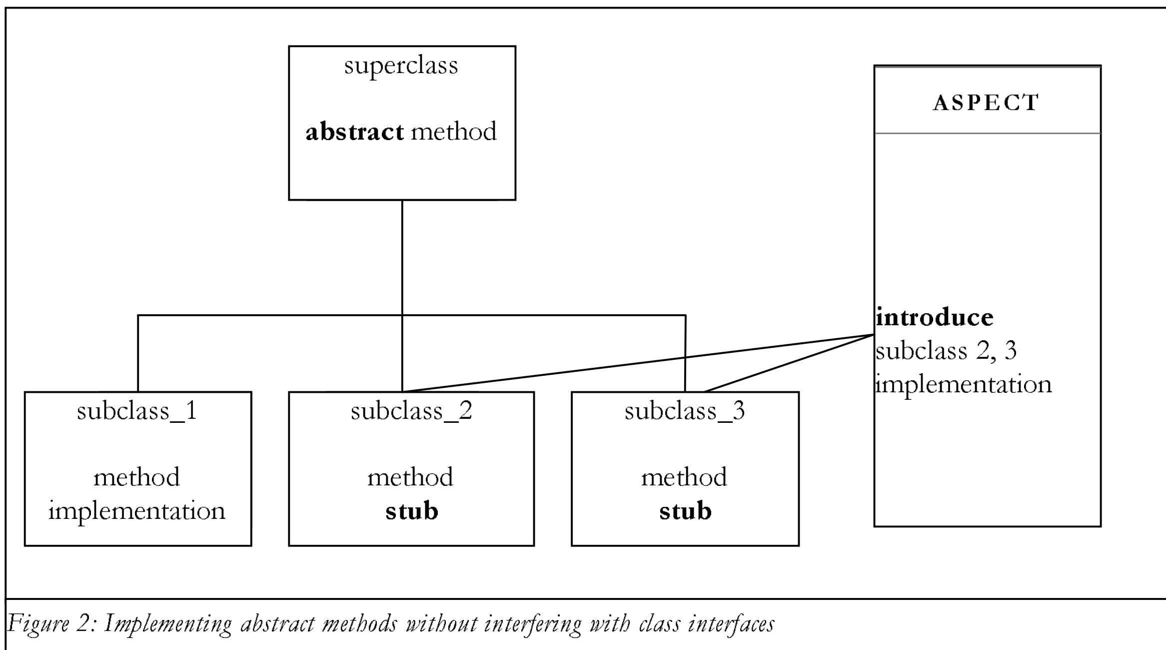

Figure 2: Implementing abstract methods without interfering with class interfaces

The following code example shows an implementation of the structure represented in the above figure.

```
class A {
  abstract void foo();
}

class B extends A {
  void foo() {}
}

class C extends A {
  void foo() {}
}

class D extends A {
  void foo() { /* D's foo implementation
                  (unrelated to B's or C's) */ }
}
```

```
aspect FooAspect {
  advise void B.foo() {
    static instead { /* B's foo implementation */ }
  }

  advise void C.foo() {
    static instead { /* C's foo implementation */ }
  }
}

--OR--

aspect FooAspect {
  introduce void B.foo() { /* B's foo implementation */ }
  introduce void C.foo() { /* C's foo implementation */ }
}
```

In the SpaceWar example that is included with AspectJ 0.3 alpha 3, there is a section of the code where all implementations of a particular abstract method are contained in an aspect. The declaration of the abstract method itself is contained in a class though. Moving the declaration of the abstract method to the aspect would be a better design decision, but the declaration of the abstract method originates in a Java AWT class. Additionally, the current development of AspectJ does not support introduce weaves for abstract methods. We attempted to move an abstract introduce weave to an aspect, but the AspectJ compiler had an internal error. When we discovered this bug, we contacted the AspectJ technical support staff and were informed that the bug that relates to introducing abstract methods will be corrected in future releases. We initially suggested a design change to SpaceWar that would move an abstract declaration to an aspect, and we received positive feedback on the suggestion. [EMa99] This was before noting that the abstract method was actually inherited from the Java AWT class. Because the abstract method is present in the signature of the superclass and cannot be removed, we now suggest a design similar to Figure 2 rather than Figure 1.

COMMENTS ON EXCEPTION HANDLING

Exception handling tends to create many issues in programming language design. These complexities occur not so much due to exception handling itself, but through its interactions with other language features. As would be expected, the introduction of AspectJ into Java creates some interesting situations. There is not much specified about AspectJ concerning how exceptions should be handled. This section will attempt to create a few guidelines for using exceptions within aspects, based upon our design philosophy.

When a method throws an exception, it is generally thought that the best place in which to handle that exception is the caller. This is why exceptions are generally propagated up the dynamic call chain, rather than the static scoping chain. Aspects create an 'exception' to this rule, however. When an aspect advises a method, it could potentially throw an exception. The calling environment is not generally suited to handle this exception because it could be something entirely unrelated to the method's intended functionality. To illustrate, let us return to the land of dragons and hunchbacks.

A hunchback is delivering a message to a mailbox with a giant division symbol painted on it. The message contents say, "What is 2 divided by 2?" The hunchback dutifully places the message in the mailbox, and awaits a response. While waiting, he falls into a deep sleep. A dragon appears, and swoops into the mailbox to retrieve the message and read its contents. She won't change the message at all, and normally would place the message back into the mailbox before the hunchback awakes. However, this time the dragon gets *really* confused while flying around with the message, and can't place the original message back in the mailbox. Instead, she panics and places a message in the mailbox that says, "I don't know what the color 'red' means." The hunchback retrieves the message from the mailbox after he wakes up, and returns the answer back to the sender. The sender looks over the message and is also thoroughly confused. He either expected an answer like the number 1, or some kind of message like, "I'm sorry, but you can't divide a number by zero." Instead, he gets this silly message about the color red. The sender of the message panics along with the dragon, and no further messages can be sent.

This example tries to point out (without referring to any language or syntax) that advice (i.e. the dragon flying with the message) should not throw error messages back to the caller. The caller is generally not equipped to handle these messages. Instead, it makes more sense if the aspect is forced to catch its own exceptions. This would be analogous to the dragon handling the error message concerning the color red, and then being able to place the original message back in the mailbox as expected. Therefore, if a throw occurs in any advice that throw needs to be caught by a catch advice. If a throw is present in an advice without a catch, this should result in a compile error.

This above suggestion solves the problem concerning exceptions mentioned in [FAJ99]. The problem in the FAQ is introduced when an exception is thrown in a piece of advice. Because the exception is not required to be caught by the advice itself, it is thrown to the calling environment. This requires that the advised method have a throws clause in its signature. While there are less than elegant ways of handling this problem, there is no support in AspectJ to handle this. We do

not think support for this should be added, however, as it would violate our principles by changing the signature of the affected method. Instead, all throws that occur in advice should be caught by an appropriate catch clause in that advice.

Some other problems occur when introduced methods throw exceptions. If a method is introduced into a class (or multiple classes), the signatures of these classes should remain the same. Therefore, these methods should be allowed to throw exceptions as part of their interfaces. This is currently not supported in the current release of AspectJ. A future release of the weaver will provide support for declaring introduced methods with throws clauses. [EMa99] It intuitively makes sense to allow this behavior, because the interface of the class does not change whether or not an introduced method throws an exception. Furthermore, the only places this exception could be thrown (given the restrictions of our design philosophy) are: back into the aspect itself (a member function or another introduced method) or into the advice section of a class' method. In either of these cases, the calling instance has scope in both the affected class and the associated aspect, providing it with as much information about the cause of the error as possible.

A slightly different case of the above occurs when the introduced method is replacing (via the **instead** mechanism, perhaps) a stub for an abstract method implementation. In this situation, the introduced method should not throw an exception, as that would change the signature of the subclass' implementation to be different from the parent's abstract method signature. However, if all instances of an abstract method are present in the aspect (i.e. no stubs, and the abstract method is introduced into the superclass) then it makes sense to allow that method to throw an exception, as every instance of that method's signature will change accordingly. This case is similar to the one described in the above paragraph.

COMMENTS ON ASPECTS AND ORTHOGONALITY

Orthogonality is stated as the principle that, "Independent functions should be controlled by independent mechanisms." [Mac87] This means that multiple language constructs should not have overlapping functionality. Complete orthogonality is difficult to achieve in programming languages without making the language overly complex and difficult to use (i.e. ALGOL68), but it is still important to maintain some level of separation between language constructs. Not everyone agrees with this particular principle [Wal99], and we also question the degree to which a programming language should strive to be orthogonal. However, we think some degree of orthogonality is necessary.

The overlapping functionality provided by aspects and classes is a clear violation of this principle. A programmer can create an aspect that behaves identically to a class. This means that there are two programming constructs with which to create classes. A mechanism for preventing this abuse of aspects was mentioned in an earlier section. If aspect instantiation is restricted so that it only can occur inside of aspect code, aspects cannot provide the same functionality as classes. This significantly reduces the amount of overlap between the two constructs.

A related question is: should aspects be permitted to inherit from classes and should classes be allowed to inherit from aspects? Currently, this is permitted. This does not make sense when the definitions of aspect and class (i.e. component) are considered. If a component is a result of the system's functional decomposition, it is itself a functioning unit. It can inherit from other functional units, but should not be allowed to inherit from aspects because they are not functional units themselves. A related issue is that there is a bug in the current implementation of AspectJ that is caused by circular inheritance. See [EMa99] for more details.

3 – Aspects and Non-Object Oriented Languages

DO APSECTS APPLY TO PROCEDURAL LANGUAGES?

In object-oriented programming, a program is partitioned into objects representing real world entities. Within this program, various aspects may be spread over several objects. Aspect oriented design was developed to deal with these crosscutting concerns in object-oriented programming languages. However, one is forced to question whether these crosscutting concerns are a direct result of object orientation, or whether crosscutting concerns exist in other programming language paradigms. This question was briefly addressed in the AOP workshop at ECOOP97. Their conclusion can be succinctly stated as follows:

> Further we can state that aspect-orientation is not bound to object-orientation only. All existing programming paradigms like procedural, functional, logical and object-oriented paradigm provide models to express real world entities. [AEC97]

Given that crosscutting concerns exist independently of the programming paradigm used, how does one identify these concerns in different paradigms? More specifically, what types of crosscutting concerns exist in procedural languages?

Not surprisingly, crosscutting concerns in procedural languages are quite similar to those in object oriented languages. Different procedures may need to share data and/or other procedures, and it may be desirable to insert additional code into certain procedures. At first glance, it appears that the first two of these crosscutting concerns could be solved using object orientation. Objects are an encapsulation of data and procedures that utilize this data. However, as in object oriented programming, there are crosscutting concerns that do not encapsulate nicely into a single object. Examples include debugging aspects, synchronization aspects, and display aspects as have been previously discussed.

Currently, within procedural languages, procedures can share data and other procedures either through parameter passing mechanisms or through a common name space. When procedures need to share a large number of items, parameter passing can become awkward. Thus, programmers will often resort to utilizing shared name spaces. In block structured languages, the name space is partitioned into a hierarchical block structure in which the name space of an outer block is visible to that block's inner blocks. Therefore, if procedures P1 and P2 wish to have access to procedure P and variable V, then P and V need to be declared either in the same block as P1 and P2, or in a block outside of it, as can be seen in the following code example:

```
procedure foo;
   begin
      var V: real;
      procedure P;
         begin
            { procedure body }
         end
      procedure P1;
         begin
            { invoke P }
            { access V }
         end
      procedure P2;
         begin
            { access V }
            { invoke P }
         end
   end
```

In the C language, name spaces are either local and visible only to a specific procedure, or global and visible to all procedures. The global name space in C can be partitioned by encapsulating components of a program into a single file and making certain variables and procedures **static** to that file. Thus, to continue our example, in the C language, P1, P2, P, and V can be placed in the same file where P and V are declared **static**:

```
static float V;
static void P();

void P1()
{
   /* access V */
   /* call P */
}

void P2()
{
   /* access V */
   /* call P */
}
```

Thus, it should be clear how block structured languages and C can use a shared name space to allow procedures P1 and P2 to access the procedure P and the variable V.

However, there are clear limitations to the scoping mechanisms in both block-structured languages and in C. In [Wul73], Wulf and Shaw identify four distinct problems with block structured languages: side effects, indiscriminate access, vulnerability, and no overlapping definitions. In block structured languages, side effects can arise from the fact that a variable defined in an outer block can be modified by a procedure defined in an inner block. Thus, it may not be apparent when that inner function is used that the variable defined in the outer block will be modified. Side effects occur in almost all programming languages and do not provide limitations in expressing crosscutting concerns. Indiscriminate access and vulnerability are problems specific to block

structured languages and the existence of these problems suggests that block structure is not an effective mechanism for expressing crosscutting concerns. The lack of overlapping definitions captures an inherent problem with block structure for expressing certain types of crosscutting concerns. The C language does not allow a nested block hierarchy, and thus does not suffer from the problems of indiscriminate access and vulnerability. However, it too fails to provide support for overlapping definitions. It is our contention that by providing an aspect construct within a block structured language or a language such as C, these problems can be solved and the limitations can be overcome.

Indiscriminate access is a fundamental problem with block structured languages. It involves the inability to prevent access to variables or procedures when that prevention is desired. The classic example of indiscriminate access involves a stack implementation, in which the stack data structure must be at least within the same block as the push and pop procedures, if not in an outer block. As a result, any procedure that wishes to access the push and pop routines will by default have access to the data structure as well. Vulnerability is the complement of indiscriminate access in that it may be impossible to *preserve* access to a variable when such preservation is desired. Vulnerability can occur when a procedure in an inner scope modifies a variable defined in an outer scope. Then if the program is modified and a new block containing a variable of the same name is inserted between the other two, then the inner procedure will no longer be accessing the correct variable. The final problem with block structured languages is the lack of provision for overlapping definitions:

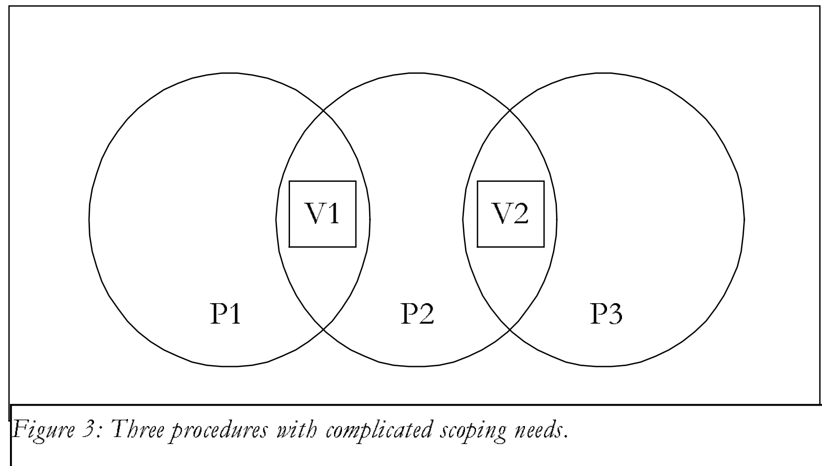

Figure 3: Three procedures with complicated scoping needs.

Suppose that we have three procedures P1, P2, and P3 that wish to share two variables V1 and V2 in the following manner: P1 and P2 share V1, P2 and P3 share V2, P3 should not be able to access V1, and P1 should not be able to access V2. It is impossible to implement this type of relationship in a block-structured language, as can be seen in the following code segment:

```
procedure overlap
  begin
      var V1: real;
      var V2: real;
      procedure P1
         begin
         { can access V1 }
         { but can also access V2! }
         end
      procedure P2
         begin
         { can access V1 }
         { can access V2 }
         end
      procedure P3
         begin
         { can access V2 }
         { but can also access V1! }
         end
      ...
  end
```

As can be seen, P1, P2 and P3 must all be placed in the same block, and V1 and V2 must be placed either in the same block as the procedures or in an outer block. In this manner, the last two restrictions of the relationship will be violated as P1 can access V2 and P3 can access V1. In this case, a crosscutting instance exists which block structure cannot express. Thus, it is clear that block structure is not an effective mechanism for capturing crosscutting concerns due to the potential programming problems it can cause (indiscriminate access and vulnerability), and its inability to capture certain types of crosscutting (overlapping definitions).

The C programming language does not manage its name space in a nested block hierarchy, and subsequently does not suffer from the problems of indiscriminate access and vulnerability. However, name space management in C does not solve the problem of the lack of overlapping definitions, as can be seen in the following example:

```
static float V1, V2;

void P1()
{
   /* can access V1 */
   /* but can also access V2! */
}
void P2()
{
   /* can access V1 */
   /* can access V2 */
}
void P3()
{
   /* can access V2 */
   /* but can also access V1! */
}
```

As can be seen, the only way to implement the relationship would be to have P1, P2, P3, V1, and V2 all in one file with V1 and V2 static to that file. However, in this case P3 can access V1 and P1 can access P2 which violates the relationship. Thus, whereas the C programming language appears to be somewhat better than a block structured language in capturing crosscutting concerns, it still cannot properly express an instance of overlapping definitions.

It is our contention that aspects can be used as a mechanism to extend the scoping rules of procedural languages to capture crosscutting concerns without raising the problems of indiscriminate access and vulnerability, as well as providing a means of expressing overlapping definitions. An aspect can be used as a container to store specific variables and procedures. Then, access to this aspect could be specified on a procedural level, meaning that only certain procedures can access a particular aspect, regardless of their nesting in a block structured language or location in a particular file in a C program. The details of instantiating an aspect and providing access to it will be discussed in a later section, as there are several complicated issues that arise. However, assuming that this can be done, it is clear how aspects can solve the three problems of indiscriminate access, vulnerability, and no overlapping definitions by providing a new name space that is only accessible by certain procedures.

By using an aspect, multiple procedures can share a variable without having to rely on nested block structure, as is indicated by the following code segment:

```
aspect A
   accessed by P1, P2;
   begin
      var V: real;
   end

procedure P1;
   begin
      { access V }
   end

procedure P2;
   begin
      { access V }
   end
```

Furthermore, access to this variable can be prevented by limiting access to the aspect to only those procedures that need to share the variable. To preserve access to the variable, the procedure would know that it is accessing a variable in the aspect and not one in an outer scope. Thus, it should be possible to prevent a procedure from accessing an incorrect variable due to name clashes, as is the case in the vulnerability problem.

Additionally, the extended scope that an aspect could provide will allow for overlapping definitions. Consider the above example of overlapping definitions in which P1 and P2 want to share V1; P2 and P3 want to share V2; but we do not want P1 to access V2 or P3 to access V1. This could be accomplished with two aspects A1 and A2, as demonstrated in the following code sample:

```
aspect A1
   accessed by P1, P2;
   begin
      var V1: real;
   end

aspect A2
   accessed by P2, P3;
   begin
      var V2: real;
   end

procedure P1;
   begin
   { can access V1 }
   { cannot access V2 }
   end

procedure P2;
   begin
   { can access V1 }
   { can access V2 }
   end

procedure P3;
   begin
   { can access V2 }
   { cannot access V1 }
   end
```

Note that aspect A1 contains V1 and is accessible by both P1 and P2, and A2 contains V2 and is accessible by both P2 and P3. In this manner, P1 cannot access V2 because it is contained in aspect A2, which is not accessible to P1. Access to V1 is similarly prevented.

At the conclusion of their paper, Wulf and Shaw identify several properties that a "suitable alternative" to block structure will possess. These include:

- The default should not be to extend the scope of a name to inner blocks
- The right to access a name should be by mutual agreement between creator and accessor
- Access rights to a structure and to its sub-structures should be decoupled [Wul73]

It is worth noting that aspects satisfy these three specific criteria. Because access is on a procedural level, if a certain procedure has access to an aspect, a procedure in an inner block will not have access to that aspect unless it is specifically given. A variable or procedure in an aspect will only be accessible to those procedures specified in the aspect, thus satisfying the second property. Finally, because sub-structures can be stored in aspects rather than at a particular scoping level, access to the structure will not imply access to its substructures.

Thus it should be clear that aspects are a *valid* alternative for capturing crosscutting concerns in procedural languages. By providing a new name space with limited access, aspects can provide more crosscutting capabilities than block structure and C scoping mechanisms, without the associated problems and limitations. However, whether aspects are a *viable* alternative is an entirely different question. The next section will discuss many of the semantic and syntactic issues associated with implementing aspects in a procedural language.

SEMANTIC AND SYNTACTIC ISSUES FOR ASPECTS IN PROCEDURAL LANGUAGES

Upon approaching Kiczales with our ideas for applying aspects to procedural languages, he replied with the following:

> I definitely agree with what you are saying! Definitely aspects can and should be applied to procedural languages. And one kind of cross-cutting structure they should be able to capture in a procedural language is the issue of sharing data across a set of the procedures. … I think its a winning idea, there are some subtle issues that will have to be worked out that should prove fun to take on. [EMa99]

We had some ideas as to what types of "subtle issues" were involved, but, we felt it was important to determine what types of issues he was referring to. In his reply, he stated:

> Well I think the scoping and state issues are going to be the trickiest. The issues will be what are the analog of aspect instances? [EMa99]

Thus, we have endeavored to determine how aspects will exist in a procedural language. Important questions we have attempted to answer are:

- Given that aspects contain variables to be shared across procedures, will the variable bindings be shared across procedures in general, or across specific procedure invocations?
- If they are to be shared across invocations, what exactly does that mean?
- How are components of aspects referred to within procedures?
- How will protection of an aspect's components be accomplished?
- How can static advising be accomplished?

Our analysis of these questions will follow and then we will discuss a possible implementation. As in the case of AspectJ, it is our belief that some of the answers to these questions will have to be determined through user feedback. Thus, we will discuss several options for some of these questions. We also include our analysis of which choices may prove to be correct.

Given that aspects contain variables to be shared across procedures, will the variable bindings be shared across procedures in general, or across specific procedure invocations?

There does not appear to be a simple answer to this question. Variables within aspects can be viewed as global to those procedures that have access to the aspect. In this case, the aspect would be instantiated once during elaboration, and the variables would be bound to the specific procedures once. Thus, the variables would remain static to the procedures throughout the execution of the program. It is not entirely clear whether this would be beneficial or not. This solution will in effect, create limited access global variables. The other option involves creating an instance of the aspect each time the procedures are invoked, resulting in a more localized binding. If this option is chosen, the behavior of these aspects will have to be carefully specified. Which option would be preferable to programmers is not clear, however, it seems that a dynamic instantiation of aspects with regards to procedure invocation is the more general case. For example, a global static instantiation of an aspect could be accomplished by having it instantiated by the main procedure in the program. Thus, since dynamic instantiation subsumes a global static instantiation, we feel that it is the correct design decision.

If bindings are to be shared across invocations, what exactly does that mean?

It may be the case that programmers will want the variables contained in an aspect to be bound and initialized on each procedure invocation. However, what exactly does this mean? Does this mean that the variables are bound when any procedure associated with the aspect is invoked? Clearly, this will not be very effective in sharing variables across procedures. Consider the following example:

```
aspect foobar accessed by foo, bar
{
   float V;
}


void foo()
{
   /*calls bar a bunch of times */
}

void bar()
{
   /*modifies V */
}


int main()
{
   /* call foo a bunch of times */
   /* what will be the value of V after each call to foo? */
}
```

Suppose that functions foo() and bar() wish to share a variable V, and that foo() calls bar(). To accomplish this, we will define an aspect foobar, that contains the variable V and is accessible by both foo() and bar(). If V is bound on each procedure invocation, then V will be re-initialized upon the call to bar(), thus preventing the procedures from sharing the variable. It seems that re-instantiating foobar each time any accessing procedure is called is not very useful.

Thus, we have decided that it will be necessary to associate an aspect with a "parent" function. In this case, a new instance of the aspect will be created upon each invocation of the "parent" function. It is assumed that the parent function will invoke any procedures that will access the aspect. In the previous example, foo() would be the parent function of foobar, so that upon each call to foo(), a new instance of foobar will be created, causing the variable V to be bound to that particular invocation of foo(). As a result, V will also be bound at this time to any procedures that foo() may call that can also access V, namely bar(). This will effectively allow the functions foo() and bar() to share the variable V while still allowing V to be bound on each invocation of foo().

How are components of aspects referred to within procedures?

This is an entirely syntactic question. To continue our example, how will the functions foo() and bar() refer to the variable V. We believe that there are two possible solutions to this problem as indicated in the following code sample:

```
aspect foobar accessed by foo, bar
{
   float V;
}

void foo()
{
   /* access V by name */
   V++;
}

--------------OR--------------

void bar()
{
   /* access V using dot notation */
   foobar.V++;
}
```

One solution is to allow them to simply use the name V. Another possibility is to refer to components of an aspect using the name of the aspect. The familiar dot notation could be used to accomplish this. In this manner, foo() and bar() would access V as foobar.V, rather than simply by the name V. We believe that referring to a component of an aspect by name alone is not the correct choice. Because procedures access aspects, rather than having aspects introduce components to procedures, it should be clear to the programmer that the variable or procedure being accessed is contained in the aspect, rather than in the current name space. Finally, by coupling the name of the component with the name of the aspect, we can greatly reduce potential

name clashes. This may seem to violate our design philosophy, but as will be discussed later, our design philosophy applies only to aspect-oriented programming over objects.

How will protection of an aspect's components be accomplished?

The core reason for adding aspects to a procedural language is to be able to express the crosscutting that can occur among procedures. To accomplish this, we need to be able to protect the components of an aspect from being accessed by other procedures that are not allowed to. In our above example, how can we prevent an evil_foo() procedure from accessing V, if it is not allowed to access the foobar aspect. Our solution is to "borrow" a familiar concept from operating systems: access control lists. The reader may have noticed the **accessed by** keywords associated with each previously defined aspect. These keywords are used to specify which procedures are able to access the aspect. Thus, within the aspect foobar, it can clearly be specified that the variable V can only be accessed by the procedures foo() and bar(), and that any other procedure (such as evil_foo()) should not be allowed to access V. One interesting design decision is whether individual components of an aspect should have their own access control lists or there should be one access control list for the entire aspect. Whereas AspectJ introduces variables and methods "one at a time," we feel that a single access control list for the entire aspect will suffice. Since the aspect is attempting to capture crosscutting concerns across a set of procedures, these procedures should be able to access everything contained in the aspect. It may prove true that programmers will wish to limit access at a component level, rather than an aspect level, however, it is our contention that the majority of the time, programmers will wish to grant access to the entire aspect.

How can static advising be accomplished?

To express crosscutting concerns in procedural languages, we have described how aspects can be used to extend the name space in these types of languages. However, further crosscutting concerns can be expressed by allowing aspects to statically advise different procedures. We feel that this can be accomplished in a similar manner to the way it is implemented in AspectJ. A pre-processor can be utilized to weave the additional code into the appropriate procedures. This will allow aspects of the program that are implemented in different procedures to potentially be encapsulated in one central mechanism.

We have given considerable thought to how aspects may be implemented in the C programming language. Many of the techniques utilized should generalize to other procedural languages. We have designed an implementation based entirely on pre-processing, with minimal run-time support necessary to implement aspects in the C language.

The simplest issue to implement for aspects within a procedural language is static advising. As previously stated, any advised code can be woven into the appropriate procedures by a pre-processor. To provide protection, access control can be implemented through the pre-processor as well. As it parses the code, it can determine whether a particular procedure is allowed to access a particular component of an aspect based on the control list, and generate an error whenever an illegal access is attempted. In this manner, the variables and functions contained within an aspect can be protected. Once it has been determined that all accesses to components of aspects are legal, we need some mechanism to make the components of the aspects visible to the procedures. Since aspects can contain both variables and functions, we should examine each one separately.

Since the pre-processor has already enforced protection, it will be safe to make the component functions of the aspect globally visible. Thus, all functions inside an aspect can be made globally visible to the entire program. To prevent name clashes and improve readability of the processed code, it may prove useful to modify the names of the functions to indicate their membership in an aspect. Consider the following example:

```
aspect foobar accessed by foo, bar
{
   int X = 13;
   int Y = 10;
   int Z;
   void fooey();
}  parent main;

void foo()
{
   foobar.fooey();
}

void bar()
{
   foobar.Y--;
   foobar.Z = foobar.X + 42;
}
```

Here the aspect foobar contains a function fooey(), which is called by the function foo(). One possible choice for fooey()'s new signature could be _foobar_fooey(). Now all references to fooey() within the program can be replaced with _foobar_fooey(), as shown here:

```
void foo()
{
   _foobar_fooey();
}
```

The final issue to be examined involves how the variables within an aspect will be handled. Since aspects are associated with a parent function, we have developed two different strategies to support this feature. Both strategies involve encapsulating the variables of an aspect into a single C structure. One way to support dynamic instantiation is to make this structure a local variable of the parent function. This will allow all of the variables to be instantiated on each invocation of the function. Then, a pointer to this structure can be passed as an extra parameter to any function that is called by the parent. The pointer will keep being passed down as those functions make additional function calls. In this manner, the structure will now be visible to all functions that are directly or indirectly called by the parent. It may be the case that certain functions that are called should not be able to access the aspect, however, since protection was already enforced by the pre-processor, it will not be a problem to pass the pointer to them. Then, all references to variables contained within the aspect can be replaced with a reference to the structure, as seen in the following example:

```
/* inserted by pre-processor */
struct _foobar
{
   int X;
   int Y;
   int Z;
}
/* end inserted by pre-processor */

int main()
{
   /* inserted by pre-processor */
   struct foobar _foobar_aspect;
   _foobar_aspect.X = 13;
   _foobar_aspect.Y = 10;
   /* end inserted by pre-processor */

   /* additional code in main*/

   foo(&_foobar_aspect);

   /* additional code in main*/

   return 0;
}

void bar(struct foobar *_foobar_aspect)
{
   /*pointer to structure containing variables passed in */
   _foobar_aspect->Y--;
   _foobar_aspect->Z = _foobar_aspect->X + 42;
}
```

This scheme will allow for dynamic instantiation of an aspect upon invocation of its parent function, and still allow the necessary procedures to access the aspect.

A second solution to this issue involves associating a stack with each aspect. Thus, upon invocation of the parent function, a pointer to a new instance of the structure will be created and pushed on the appropriate stack. This stack will be globally visible to the entire program; however, since access control has already been enforced, this is not a violation of the protection scheme. Then, when a procedure references a variable contained in an aspect, it will simply access the instance of the structure that is on top of the stack. This is shown in the following code segment:

```
/* inserted by pre-processor */
struct _foobar
{
   int X;
   int Y;
   int Z;
}
/* end inserted by pre-processor */

int main()
{
   /* inserted by pre-processor */
   struct _foobar _foobar_aspect;
   _foobar_aspect.X = 13;
   _foobar_aspect.Y = 10;
   _aspect_push(&_foobar_aspect,_foobar_stack);
   /* end inserted by pre-processor */

   /* additional code in main */

   foo();

   /* additional code in main */

   /* inserted by pre-processor */
   _aspect_pop();
   /* end inserted by pre-processor */

   return 0;
}


void bar()
{
   struct foobar *_foobar_aspect;
   /* accessing top of aspect stack to retrieve instance */
   _foobar_aspect = (struct foobar *)_aspect_top(_foobar_stack);
   _foobar_aspect->Y++;
   _foobar_aspect->Z = aspect->X + 42;
}
```

In this manner, any procedure that needs to access the aspin may do so by accessing the globally visible stack. The aspin is popped off of the stack upon termination of the parent function.

It is unclear which implementation will provide better run time performance, however they both appear to provide the necessary functionality.

The above-described pre-processor should enable aspects to be implemented for the C programming language. It provides capability for static weaving, access control for components of the aspect, and dynamic instantiation of aspects with respect to a parent function, by implementing each of these features in standard C code. The processed code can then be sent through a standard C compiler to produce the program. The only additional run time overhead will result from accessing the variables in an aspect, either by passing the pointer or by accessing the top of the aspect stack.

DISCUSSION OF ASPECTS IN PROCEDURAL LANGUAGES

It is clear that aspects play a different role in procedural languages than they do in object-oriented languages. We feel this is a result of the fundamental differences between the paradigms. As previously stated, a component of a program is a modular unit of functional decomposition, which addresses a specific concern or function of the program. Within an object-oriented language, a component is implemented as an object whereas in a procedural language, a component is expressed as a procedure.

The fundamental basis for our aspect design philosophy for object-oriented languages is that an object is a "something" (Philosophical point 1). Starting from there, we have attempted to separate objects and aspects. However, if an object is a "something," then a procedure can be viewed as "part of something," and as such, should not be completely orthogonal to aspects. Thus, it is apparent that our design philosophy for co-composing aspects and objects does not have a direct corollary in procedural languages. For example, procedures are both aware of and dependent on aspects. Not only do procedural hunchbacks see the dragons, but they occasionally need their explicit help to get where they are going.

4 – Summary of Suggestions

1. The term **aspin** should be introduced as an accepted abbreviation for "aspect instance."

2. Apect instantiation and control of the aspin's domain should be limited to code originating in the aspect.

3. Aspects should be permitted to view class private elements, but not permitted to change them without going through the same interfaces available to other classes (preserving encapsulation).

4. Abstract methods that are part of the signature of a superclass must remain part of the signature of a subclass.

5. Abstract methods that are introduced to a superclass by an aspect should also be implemented by an aspect for all subclasses that implement the abstract method. This introduction does not affect the signatures of the classes.

6. To support an aspect that implements a subset of the abstract implementations, we propose *instead* weaves to be implemented either with the addition of an **instead** keyword or the addition of an overwriting property of the **introduce** weave. In both cases, the abstract method's signature must be present in the subclass, while the implementation is contained in the aspect.

7. Throwing an exception in a piece of advice changes the interface of the advised method. Therefore, we suggest that any exceptions thrown in advice should also be caught in advice.

8. Classes and aspects should be orthogonal concepts as much as possible. They should not be able to inherit from each other.

9. Access control lists should be available in aspects as a language construct.

10. Aspects should be applied to programming paradigms other than OO.

11. The dragon and hunchback illustration used throughout this paper should be accepted as the canonical analogy to describe Aspect Oriented Programming. Furthermore, O'Reilly Publishers should create a book entitled *Aspects in a Nutshell* that has a dragon and a hunchback on the cover.

5 – Concluding Remarks

We believe that Aspect Oriented Programming is a significant step forward in the evolution of programming languages. Specifically, we recognize that the modularization of crosscutting concerns is of utmost importance in constructing readable, complete and maintainable code. Object orientation provides a solid foundation upon which to model the functional decomposition of many problem domains into components, which can be arbitrarily composed with each other. AOP provides a mechanism for encapsulating crosscutting concerns into modules that can be co-composed with the components that OO provides. We have also discussed how AOP can be applied to other programming paradigms as well, where the components are not necessarily objects.

Our philosophy of design grew out of an attempt to balance a number of generally accepted design principles that tend to work against each other. In particular we want to enable aspects to do their job of providing abstraction and automation, while still maintaining the principles of information hiding and manifest interface. We feel that the decision to grant a large degree of freedom to AspectJ's user base is a good idea. We understand that many of the decisions concerning language features have often been left up to user feedback and actual use cases. We present these guidelines in the hopes that the AOP community will move towards this more restricted view of how aspects should be used. This is to prevent users from maliciously using language features to violate traditional programming principles. Aspects are principally useful for improving maintainability and should not be abused solely to improve writability.

Our suggested changes are based primarily upon the principles stated in our Philosophy of Aspect-Oriented Programming. The changes are not terribly complicated, but they do represent significant design changes from the current implementation. These are summarized in section 4.

It is May of 1999. The dragons are running wild. They tear giant holes in walls. They call them doors. They shatter the glass ceilings. They fly in at will. The hunchbacks are running scared. The sanctity of the home must be restored. The wizards must cast a spell. The dragons will be tamed.

Special thanks to Megan Pealer of Lehigh University for her assistance in constructing and proofreading this document.

Special thanks to Gregor Kiczales and Jim Hugunin of Xerox PARC for their invaluable assistance.

Bibliography

[AEC97] Lopes C. V., Mens K., Tekinerdogan B., Kiczales G., *Aspect-Oriented Programming Workshop Report*, ECOOP97 Proceedings, 1997.

[Aks98] Aksit M. and Tekinerdogan B., *Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters*, ECOOP98 Proceedings, 1998.

[AOP99] Kiczales G., *Aspect-Oriented Programming*, slides from talk given at University of Virginia, 1999.

[AsJ99] Xerox PARC, *The AspectJ Primer*, http://www.parc.xerox.com/spl/projects/aop/aspectj/primer

[Bar98] Bardou D., *Roles, Subjects and Aspects: How do they relate?*, ECOOP98 Proceedings, 1998.

[Bro95] Brooks F., *The Mythical Man-Month*, Addison-Wesley, Reading, MA. 1995.

[Cza97] Czarnecki K., Eisenecker U., Steyaert P., *Beyond Objects: Generative Programming*, ECOOP97 Proceedings, 1997.

[EMa99] Xerox PARC Email Correspondences, 1999. (See Appendix)

[FAJ99] Xerox PARC, *AspectJ FAQ*, http://www.parc.xerox.com/spl/projects/aop/aspectj/faq-aspectj.shtml

[FAO99] Xerox PARC, *Frequently Asked Questions about AOP*, http://www.parc.xerox.com/spl/projects/aop/faq-aop.html

[Ken98] Kenens P., et al., *An AOP Case with Static and Dynamic Aspects*, ECOOP98 Proceedings, 1998.

[Kic97] Kiczales G., Lamping J., Mendhekar A., et al., *Aspect-Oriented Programming*, Xerox PARC, Palo Alto, CA. June, 1997.

[LoK97] 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.

[LoK98] Lopes C. V. and Kiczales G., *Recent Developments in AspectJ*, ECOOP98 Proceedings, 1998.

[Lop97] Lopes C. V., *D: A Language Framework for Distributed Programming*, Xerox PARC, Palo Alto, CA. November, 1997.

[Mac87] MacLennan, B., *Principles of Programming Languages: Design, Evaluation, and Implementation*, Oxford: NY, 1987.

[Mat97] Matthijs F., et al., *Aspects should not die*, ECOOP97 Proceedings, 1997.

[Men97] 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.

[Mod92] Mody, R.P., *Functional programming is not Self-modifying Code*, SIGPLAN Notices, vol. 27, No. 11, November, 1992.

[Neb98] Nebbe R., *Coordination and Composition:* The Two Paradigms Underlying AOP?, ECOOP98 Proceedings, 1998.

[Oss98] Ossher H., *Operation-Level Composition: A Case in (Join) Point*, ICSE98 Proceedings, 1998.

[Wal99] Wall, L., *The Origin of the Camel Lot in the Breakdown of the Bilingual Unix*, Communications of the ACM, Vol. 42, No. 4, April, 1999.

[Weg76] Wegner P., *Programming Languages – The First 25 Years*, IEEE Transactions on Computers, Vol. C-25, No. 12, December, 1976.

[Wul73] Wulf B. and Shaw M., *Global Variables Considered Harmful*, SIGPLAN Notices, vol. 8, No. 2, February, 1972.

Appendix – Correspondences with Xerox PARC

From	To	Date	Time	Subject
Group	Kiczales, G.	4/15/99	11:11AM	AOP and mixin
Kiczales, G.	Group	4/15/99	11:42AM	RE: AOP and mixin
Group	Kiczales, G.	4/22/99	2:42PM	Question regarding AOP and Procedural Languages
Kiczales, G.	Group	4/22/99	9:08PM	RE: Question regarding AOP and Procedural Languages
Group	Kiczales, G.	4/22/99	9:57PM	AOP and "reaching in"
Kiczales, G.	Group	4/22/99	10:17PM	RE: AOP and "reaching in"
Group	Kiczales, G.	4/23/99	9:44AM	RE: AOP and "reaching in"
Group	Kiczales, G.	4/28/99	1:33PM	RE: Question regarding AOP and Procedural Languages
Kiczales, G.	Group	4/28/99	2:09PM	RE: Question regarding AOP and Procedural Languages
Group	Kiczales, G.	4/29/99	4:33AM	RE: AOP and Aspect Instances
Group	Kiczales, G.	4/29/99	4:28PM	RE: Question regarding AOP and Procedural Languages
Kiczales, G.	Group	4/29/99	5:33PM	RE: Question regarding AOP and Procedural Languages
Group	Kiczales, G.	4/29/99	7:15PM	AspectJ and Exception Handling
Kiczales, G.	Group	4/29/99	8:23PM	spec
Group	Kiczales, G.	4/30/99	12:19PM	RE: Question regarding AOP and Procedural Languages
Group	AspectJ Support	5/2/99	11:42AM	Introducing Abstracts
Hugunin, J.	Group	5/3/99	1:03PM	RE: Introducing Abstracts
Group	AspectJ Support	5/4/99	6:53PM	Introducing a method that Throws an exception (Bug report)
Group	Kiczales, G.	5/4/99	7:27PM	RE: spec
Kiczales, G.	Group	5/4/99	8:26PM	RE: spec
Hugunin, J.	Group	5/4/99	8:26PM	RE: Introducing a method that Throws an exception (Bug report)
Group	AspectJ Support	5/6/99	3:07PM	Bug with cyclic inheritance