University Of California, Irvine

School of Information and Computer Science

ICS 221 Term Paper

Trends For Separation Of Concerns Using Aspect Oriented Programming

by

Sushil K Bajracharya

Fall 2003

Course Instructor: Prof. David Redmiles

Trends For Separation Of Concerns Using Aspect Oriented Programming

Sushil K Bajracharya

School of Information and Computer Science

University of California, Irvine

sbajrach@ics.uci.edu

12th December 2003

Abstract

Aspect Oriented Programming (AOP) is widely being adopted by the current programming community as an ingenious way to separate crosscutting concerns in modular programs. Popular implementations for AOP are generic in nature and have been fruitful in addressing many common cross cutting concerns like tracing and logging. Object Oriented Programming has been quite successful in addressing issues like abstraction and information hiding that made component based software development a possibility but it suffers from the limitation of not being able to encapsulate the crosscutting issues in software as addressed by AOP. This makes AOP an advanced technique for *separation of concerns* that complements techniques like object oriented programming.

In this survey paper I present, a brief history of AOP, the current state of AOP by looking into the available features for separation of concerns in some of the current

frameworks for AOP viz AspectJ, AspectWerkz, JBoss AOP framework and Hyper/J [1]and a comparative summary of these implementations. Finally I conclude with some of my findings about the research directions for AOP as envisioned by the champions in the field.

1 Introduction

Almost always software of considerable size and business value is an evolving product. The size and the complexity of the software and the interdependency of modular components that make up the software as a whole increases as the software evolve. Software engineers at their disposal have design guidelines like abstraction, information hiding and modularity that attributes to high cohesion and low coupling of components eventually aiming to produce a tractable and flexible software product. These design principles are based on concepts like *separation of concerns* and *functional decomposition* pioneered by the grandmasters in the field of software like Dijkstra [Dij76] and Parnas [Par72]. These concepts have heavily dominated the programming methodology we have adopted and directed the way we produce software today.

Even with these design tools at disposal there remain subtle implementation issues that bind these components together in a cohesive whole. These issues (such as concurrency and memory access patterns [KLM+97], especially perceptible during runtime) are not prone to the conventional decomposition techniques. These issues *crosscut* software components and stand out as the *properties that affect the performance or semantics of the components in systematic ways* [KLM+97]. AOP brings into light a new perspective of problem decomposition (or, arguably composition) and software organization to address such cross cutting

[1]All of these widespread implementations are based on the Java programming language. There are AOP frameworks available for other languages too but I have restricted my study among these Java based frameworks only, both for ease and consistency in comparison.

issues.

This survey paper is an attempt to trace the evolution of AOP as a new technique for advance separation of concerns. It is organized into three parts. First, section 2 presents the history of AOP with a summary of pertinent methodologies that were influential in shaping and defining the first AOP framework. Second, section 3 discusses the present state of AOP by first looking into AspectJ as the first generic framework for AOP. Section 3.4 is a listing of some alternate AOP frameworks that are available today. It gives a cursory overview of how these frameworks address the fundamental AOP principles differently. A summary comparing these AOP frameworks is given in section 3.5. Finally, current topics in, and future directions for research in AOP are discussed in section 5.

2 Background And History

[KLM$^+$97] is *the* seminal paper that puts together the fundamental concepts of AOP. [HL95] gives a listing of common techniques for separation of concerns that were available during the year 1994, these techniques can be termed as the influential factors for the birth of AOP. [Lop02] is another excellent resource that gives a personal and insightful view on the origin of AOP by one of the original inventors of AOP. Most of the materials in this section are based on these documents.

2.1 Before AOP

Before AOP was conceived as a new programming technique there existed similar metaphors for achieving separation of concerns, though restricted in smaller domains. Those techniques were in a way or another extensions to object oriented programming and are listed below.

2.1.1 Adaptive Programming [Lie96]

Also known as the Demeter method, adaptive object oriented programming or pattern oriented programming, this technique can be thought of as an extreme elaboration of the Visitor design pattern [GHJV95] combined with flexible compositional rules for classes. Adaptive programming tries to add yet another level of abstraction upon object oriented programming by separating the tight coupling of data structures (classes) with the operations (methods) that will be performed on those data structures. For this it defines the class structure (*structural block*) in a class graph and also defines a set of patterns (*propagation patterns*) that define different ways to traverse through the classes in the graph to perform different tasks. These patterns and class graphs can be put together and compiled to an object oriented program using a pattern compiler.

Adaptive programming was used to separate concerns like synchronization and remote parameter passing by defining *transportation* and *synchronization patterns* [LL94].

2.1.2 Meta Level Programming [Kic92], [GKB91]

Before AOP was conceived at Xerox PARC people were working on a technique known as *open implementation* that was based on meta level programming and meta object protocol. The motivation behind meta level programming (based on a meta object protocol) in the open implementation project was the fact that classical modularization techniques often take away the efficiency in implementation and they constrain programmers within such abstraction boundaries.

Meta level programming gives programmer the flexibility to access the innards of language implementation by extending and presenting the implementation of the language itself as a set of meta objects that programmers can access and use to get a fine grained control over the individual program elements. Meta level programming works on the basis of the meta object protocol that governs the overall semantics of dissecting the implementation and viewing it at a meta-level that gives the programmer the pliable reflective view of the program that makes possible the dynamic runtime manipulation and control of the program flow as desired.

By programming the meta objects to intercept communication between the base objects, specific concerns can be implemented at the meta-level without disturbing the code written at the base level.

2.1.3 Composition Filters [Ber94], [AWB$^+$93]

Composition filters enhances the object model by wrapping the object with a set of *filters* that intercept all the messages that come in and go out from the object. This makes it possible to separate concerns like *real time constraints* from the classes and implement them as *filters*. Moreover such filters can be stacked together and based on conditional rules, message passing through filters can be used to realize concerns like real time constraints.

2.1.4 Subject Oriented Programming [HO93]

Subject Oriented Programming adds another dimension in software composition. It makes the notion of abstraction higher by giving a specification technique that allows the programmers to define more than one abstract view for a set of classes in an object oriented program. Each view defines the classes in terms of different behaviors of interest or in terms of an appropriate context being considered by the programmer. This is achieved by defining a *subject* that is a collection of related set of classes. Subjects in themselves can be self-complete or can act as a part of a system to be built. Subject Oriented Programming makes it possible to group or *compose* these subjects together using some *composition rules* without modifying or accessing the source code. Subject Oriented Programming is a precursor to the next level of separation of concern technique called the *Multidimensional*

Separtaion of Concerns or the *Hyperspace* approach that is discussed in section 3.4.3

By allowing composition of same set of classes in different ways based on *declarative composition rules*, subject oriented programming allows a developer to use any combination of the available *behaviors*.

Although these projects predate AOP, they still are active and being carried on. Some of the recent works in these areas that contribute to AOP techniques are discussed in section 4.1.

2.2 The Birth Of AOP

Among the four techniques for separation of concerns listed in section 2.1 the first two had a profound effect on the birth of AOP. Crista Lopes who had continued her work on a language framework called DJ [Lop97] (Distributed Java) that had roots linked to Adaptive Programming was one of the co-inventors of AOP who joined the team at Xerox PARC led by Gregor Kiczales working on a project called *open implementation*. Metaobject protocols and reflection [KLM$^+$97] was used in prototyping earlier AOP systems and elements of DJ [Lop02] were used to define a higher level *aspect language* and a *weaver*. [2]

As already mentioned the original idea of AOP is documented in [KLM$^+$97].

2.2.1 The Fundamental Theory

This section describes the basic concept behind AOP and is based on the earlier work on AOP by the original inventors.

The principal goal of AOP was (and is) to provide mechanisms to allow separation of crosscutting *aspects* from *components* in a system. Aspects are the issues that cannot be cleanly encapsulated as a program unit whereas components can be modelled as separate

[2] Aspect language and weaver are essential components of AOP and are discussed in the preceding section.

logical entities (like classes in object oriented programs). This separation results in an easy to understand and an efficient program. To achieve this, the framework for AOP comprises of following components -

Component language A high level language with constructs like procedures, classes etc to program the components. Component language can be transformed to produce *component program*.

Aspect Language An extension to the component language or a separate language to program the aspects. Aspect language gives *aspect program* when implemented.

Aspect weaver This is a tool that processes the component and the aspect language (precisely the programs written in these languages) and combines them together to produce the final executable code.

Few important points should be noted regarding this framework for AOP -

- Component programs are simple as they cannot *preempt* anything aspect programs need to control. For example, if Java is being used as a component program and if concurrency is one of the aspects then using keywords like `synchronized` and methods like `wait()` should not be allowed in the component language. This should rather be addressed by the aspect language.

- The aspect weaver does not work as an optimizing compiler in producing efficient code by combining the component and aspect language. All the possibilities for optimizations should be already represented by the aspect language in the aspect program. The weaver simply integrates programs written in component and aspect languages.

- Weavers take into account the presence of *join points* during the *weaving* process. Join points are elements of the component language semantics that the aspect programs coordinate with. A simplest example might be the point of method call in a program or a point in a program that can be captured to implement a technique like loop fusion.

[KLM$^+$97] presents two important motivations behind AOP -

Program Understanding Aspect language by taking out cross cutting concerns from the main program code should make the overall program more expressive and easier to comprehend.

Efficiency In general high level languages the modifications that are made to an existing code to enhance the performance often results in a more tangled code and thereby results in an efficient but obscure code. With AOP the aspect language should be able to encapsulate the *smart* logic for optimizations (mainly, domain specific optimizations) and the tangling should get out of the picture by delegating the task to the weaver.

2.2.2 A Simple Example

Lets assume that we need to trace and log exceptions as *aspects* in a program such that whenever there is an exception it must be recorded in a log file. One way to achieve this functionality would be to include a statement or a method call in the program everywhere it does exception handling such that the statement or method would write the error details in the log. In such a scenario it would be the programmers responsibility to add that particular piece of code in all the places in the program where errors may occur (like in every exception handler). This definitely cross-cuts the entire program, as error handling might be required in any place in the program no matter what modularity exists among components.

Rewriting such programs with an AOP framework would entail capturing all the references to exception handling in a separate *aspect code* such that the *weaver* would compose the *aspect code* that implements exception logging with the *program code* that does not need to know that logging is being enabled. In this scenario a *join point* might be represented by a *method call* [3] or more precisely a *method call pertaining to exception handling*. The weaver then recognizes such calls as join points (which of course is expressed in an aspect language) in the program code and weaves it with aspect code to produce a final executable code that will achieve the task of exception logging.

So, the logging issue that would have been scattered around the entire program, hooked with all the exception handler code can be represented as a single aspect and easily maintained with an appropriate AOP framework.

2.3 The Transition

When AOP was conceived there were two possible implementation strategies first, a domain specific AOP for dealing with issues like code efficiency that would have a richer and fine grained model for join points and second, a general purpose AOP that could be used for dealing with common crosscutting issues in program.

The first implementation of a general purpose AOP framework gave birth to AspectJ. That was a significant transition for AOP from a concern specific implementation with DJ (that handles issues like synchronization and remote parameter passing) as its core to a general purpose framework with AspectJ as its aspect language [4].

Different variants of AOP and their transitions is documented in [Lop02] and [LK98].

[3]It should be noted that frameworks like AspectJ has a richer join point model that does not require to have method calls as join points for exceptions, method call is mentioned here just for simplicity

[4]At this point its appropriate to denote AspectJ as an aspect language, but it would be equally appropriate to refer to AspectJ as an AOP framework. AspectJ refers to both the AspectJ AOP framework and the AspectJ aspect language in the AspectJ AOP framework. It is easy to identify the implication depending upon the context.

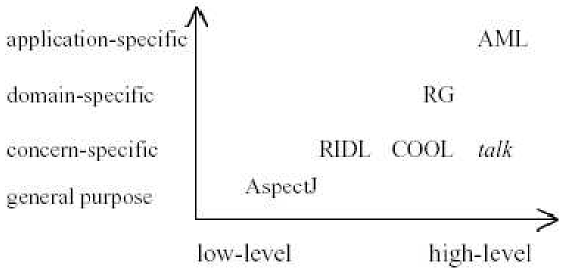

Fig. 1. Range of AOP languages' features in some of our work. RIDL is the remote data transfer language in [5]; COOL is the synchronization language in [5]; RG is the image processing system in [6]; AML is the sparse matrix language in [1]; "*talk*" refers to the language in the slides of the invited talk [2]

Figure 1: Figure from [LK98] showing transition among different AOP variants.

Figure 1 shows a graph taken from [LK98] that depicts the transition among different variants of AOP. For more details I recommend referring these documents [Lop02] and [LK98].

3 AspectJ - The Rise Of Generic AOP

After the release of an early version of AspectJ in 1998 it received a wide range of support all over, especially from the object-oriented community. Since then AspectJ has evolved a lot and many additional features have been added to the aspect language itself. AspectJ has moved out from Xerox PARC and is now an open source initiative as a part of the Eclipse project [WEBa]. AspectJ is at the latest version of 1.1.1 to this date and is hosted at `http://www.aspectj.org`. [WEBb].

3.1 Fundamental Concepts In AspectJ AOP Framework

AspectJ is the first generic framework for AOP and it constitutes a standard set of elements for an AOP framework. It is now a de facto standard for general purpose AOP frameworks against which other AOP frameworks are compared.

In a nutshell the current version of AOP is comprised of the following elements.

3.1.1 Component Language

Since its conception AspectJ had been using the Java programming language as its component (or base) language and no change has been made in this till today.

3.1.2 Aspect Language

The aspect language named as AspectJ itself is what adds the expressive capability to the component language Java. A full description of the AspectJ is beyond the scope of this paper. For this purpose it is recommended that readers refer to [WEBc]. Descriptions and examples in this section are based on the contents available in [WEBc].

In brief, AspectJ (aspect language) can be described in terms of the following constructs that constitute it -

Join points AspectJ now comes with a more powerful join point model that describes several join points besides method calls, reception or execution. As mentioned earlier, join point is a well defined point in an execution of a program. The strength of an AOP framework is governed by how fine grained its join point model is. AspectJ does not allow an arbitrary representation of the cross-cutting issues in a program as *aspects*. In fact only those issues that cross-cut the join points defined in AspectJ can be modelled as *aspects*.

pointcuts	meaning
call (*Foo.new(..)*)	a **call** to any constructor of *Foo*
execution(* *Foo.*(..) throws IOException*)	the **execution** of any method of *Foo* that is declared to throw *IOException* (note the use of wildcard * here)
set (*!private * Point.**)	when *any non-private* (implied by *!*) field of *Point* is **assigned**
handler (*IOException+*)	when an *IOException* or its *subtype* (implied by +)is **handled** with a catch block

Table 1: Some of the pointcuts available in AspectJ

[WEBc] lists eleven join points for the latest AspectJ release and describes all the join points and their semantics in greater detail. Some of them are **method call**, **constructor call**, **constructor execution**, **field reference**, **field set** and **handler execution**.

Pointcuts Pointcuts are constructs in AspectJ that selectively allow to choose certain join points in the program flow. AspectJ defines a set of seventeen primitive pointcuts that allow all the basic join points to be captured. Furthermore, new pointcuts can be defined on the basis of old pointcuts, and pointcuts allow use of wildcards and logical operators like *&&* (meaning logical *and*), *!* (meaning logical *not*) and || (meaning logical *or*) while selecting and specifying join points.

Table 1 lists few examples of pointcuts in AspectJ. It should be noted that **call**, **execution**, **set** and **handler** are the base pointcuts that AspectJ defines.

Advices Advices define a body of code that is executed when a pointcut is reached. An Advice binds together a pointcut (that pick up the join points) and a body of code (Java code) to run at each of those join points. AspectJ defines three kinds of advices-

1. *Before*: Before advices run upon reaching the join points and before the program continues with the join points.

```
aspect SimpleTracing {
  pointcut tracedCall():
    call(void FigureElement.draw(GraphicsContext));

  before():  tracedCall() {
     System.out.println("Entering:  " + thisJoinPoint);
   }
}
```

Figure 2: A simple example of tracing aspect taken from [WEBc]

2. *After*: After advices run upon reaching the join points and after the program continues with the join points.

3. *Around*: Around advice runs in place of the join point it operates over, rather than before or after it.

Aspects Aspects are special programming constructs that cleanly encapsulate the cross cutting concerns in separate units. They are similar to the notion of classes in the component program. Aspects are combinations of pointcuts and advices.

An example of a simple tracing aspect is given in Figure 2. This example defines an *aspect* `SimpleTracing` that contains a definition for a *pointcut* `tracedCall` that points to the *join point* of method call for method `draw` defined in the class `FiguredElement` and that takes an argument of type `GraphicsContext`. It uses a *before advice* on `tracedCall` to say that a message will be displayed every time before the method pointed by the *pointcut* `tracedCall` is executed. It has a special variable `thisJoinPoint` description of which is skipped here.

Introduction or Intertype Declarations Intertype declarations is a powerful feature in AspectJ that can extend the structure of the classes in the component code. Aspects can add attributes or can force the classes to implement certain interfaces or to

```
class Point {
   int x, y;

   public void setX(int x) { this.x = x; }
   public void setY(int y) { this.y = y; }

   public static void main(String[] args) {
       Point p = new Point();
      p.setX(3); p.setY(333);
    }
}
```

Figure 3: A class that will be extended by Introductions shown in Figure 4. [WEBc]

inherit from other classes.

Figure 3 is a listing for a simple *class* **`Point`**. Figure 4 shows how an *aspect* named **`PointAssertions`** can be defined that can extend the class **`Point`** by adding two methods **`assertX`** and **`assertY`**. These methods are declared `private` so they'll be visible within the scope of the *aspect* **`PointAssertions`** only. The original **`Point`** class will not notice that it has been forced to add two methods within the scope of the *aspect* **`PointAssertions`**. The *aspect* **`PointAssertions`** further defines two *before advices* that uses three primitive *pointcuts* each. These *advices* make the newly added methods work as guard calls to **`setX`** and **`setY`** methods of the **`Point`** class. This is an example how cleanly assertion logic can be added to a class without even modifying it, another example of separation of concerns!

3.1.3 Weaver

Another important component of the AspectJ AOP is the weaver that comes as the `ajc` compiler. This compiler takes Java (component) code and the aspect code (written in the

```
aspect PointAssertions {
   private boolean Point.assertX(int x) {
      return (x <= 100 && x >= 0);
    }
   private boolean Point.assertY(int y) {
       return (y <= 100 && y >= 0);
    }

   before(Point p, int x):  target(p) && args(x) && call(void setX(int)) {
       if (!p.assertX(x)) {
         System.out.println("Illegal value for x"); return;
       }
    }
   before(Point p, int y):  target(p) && args(y) && call(void setY(int)) {
      if (!p.assertY(y)) {
          System.out.println("Illegal value for y"); return;
       }
    }
    }
```

Figure 4: Aspect with inter-type declaration that extends the class in Figure 3 to add assertion methods. [WEBc]

AspectJ aspect language) as inputs and weaves them together to produce bytecodes that can be loaded and exectued in a Java Virtual Machine. `ajc` (that comes with AspectJ 1.1.1) can directly weave compiled bytecodes with aspect code so it is not necessary to have the source code for component code available. Detail descriptions in using ajc can be found in [WEBd].

3.2 Categories Of Aspects

In [WEBc], aspects are broadly categorized into two groups -

1. *Development Aspects*: These aspects model concerns that are important during development and debugging phase. Such concerns can be easily removed from the program after development. Development aspects take into consideration issues like Tracing, Profiling and Logging, Pre- and Post- condition checking, Contract enforce-

ment and configuration management. These issues need not be included in the final build of the software that will be distributed to the customers.

2. *Production Aspects*: These aspects model system concerns that are actually shipped with the production version of the software. Aspects under this category perform or fulfill some of the system requirements such as Caching, Transaction management, Authentication etc. JBoss server mentioned in section 3.4.1 ships its server features as system-level aspects.

3.3 Tool Support For AspectJ

Many Integrated Development Environments (IDEs) have support for the AspectJ framework. One of the oldest tool is the Aspect Browser [WEBd] that serves two purposes. It can be used as a visual interface for the `ajc` compiler as well as it can be used to visualize the crosscutting structures in the program. AspectJ plug-ins are available for popular IDEs like Eclipse, JBuilder, Emacs and Netbeans. [Ker] gives a brief overview of tool support available for AspectJ.

3.4 Alternative AOP Implementations

AspectJ has a rich aspect language and has complex techniques to deal with advance separation of concerns. This also makes it a heavy-weight framework compared to others. Alternative implementations for AOP could be viable in certain conditions when one does not need the full features of AspectJ. This section lists few of those alternatives available to AspectJ.

3.4.1 JBoss AOP

JBoss AOP framework [WEBe] comes as a part of the JBoss Java Application Server [WEBf]. JBoss AOP can be used as a standalone AOP framework as well as it can be used along with the JBoss application server. One of the significant benefits of JBoss AOP using with the JBoss server is that JBoss server ships its system-level services such as Transactions, Remoting, Clustered Remoting and Transactional Locking as aspects that are written on top of the framework.

Differences The significant differences between JBoss AOP and AspectJ can be listed as follows -

- JBoss primarily uses XML as its aspect language to model AOP constructs like Introductions, and pointcuts.
- JBoss uses interceptors (or interceptor classes) that work as the advice construct in AspectJ. Interceptor classes can be used to intercept method invocations, constructor invocations and field access.
- JBoss AOP adds the possibility of declarative programming in Java by allowing metadata to be defined as aspects in XML files that can be used to add attributes to classes and other elements of Java.
- JBoss AOP uses bytecode manipulation to attach the interceptors to the component code and uses its own class loader as it does runtime weaving.
- JBoss provides a tool called AOP Management Console with web based interface that allows to view classes that have been loaded by its classloader and being instrumented in the AOP framework. It even shows a static trace of interceptor chains, introductions, interceptor pointcuts and metadata attached to the loaded classes.

```
<?xml version="1.0" encoding="UTF-8">
<aop>
   <interceptor-pointcut class="POJO">
      <interceptors>
         <interceptor class="TracingInterceptor" />
      </interceptors>
   </interceptor-pointcut>
</aop>
```

Figure 5: Pointcut definitions in JBoss AOP using XML. [BB03]

- Above all, the major difference between the JBoss AOP and AspectJ lies in the weaving process. AspectJ can be viewed as a statically weaving AOP framework that either takes Java sources or complied classes and combine them together giving classes that can be loaded and executed in the Java Virtual Machine. With JBoss the process is more dynamic. JBoss AOP uses its own class loader to combine the Java code and instrument it by reading the XML representation of the aspect code. Then it uses a meta-object protocol based library for modifying bytecodes called *Javassist* [WEBg] to monitor all the running classes.

Examples Figure 5 shows an XML snippet that defines a *pointcut* that combines the `TracingInterceptor` class (that is the *interceptor* class and has the definition for the *advices*) with the `POJO` class (simple Java class or component code). Similarly, Figure 6 shows how *metadata* or *atributes* can be added to the java classes that belong to the component code. It says that all the `get` and `set` methods, and the `main` method in the `POJO` class will have an *attribute* named `filter` whose value is set to `true` by default.

[BB03] puts the basic concepts of JBoss AOP together with nice examples like these.

```
<?xml version="1.0" encoding="UTF-8">
<aop>
   <class-metadata group="tracing" class="POJO">
      <method name="(get.*)|(set.*)">
         <filter>true</filter>
      </method>
      <method name="main">
         <filter>true</filter>
   </method>
    </class-metadata>
</aop>
```

Figure 6: Metadata definitions in JBoss AOP. [BB03]

3.4.2 AspectWerkz

[WEBh] gives the following description for AspectWerkz -

"AspectWerkz is a dynamic, lightweight and high-performant AOP/AOSD framework for Java.

AspectWerkz utilizes runtime bytecode modification to weave your classes at runtime. It hooks in and weaves classes loaded by any class loader except the bootstrap class loader. It has a rich join point model. Aspects, advices and introductions are written in plain Java and your target classes can be regular POJOs. You have the possibility to add, remove and re-structure advices as well as swapping the implementation of your introductions at runtime. Your aspects can be defined using either an XML definition file or using Runtime Attributes.

AspectWerkz offers both power and simplicity and will help you to easily integrate AOP in both new and existing projects."

Differences AspectWerkz has basically the same model for AOP as AspectJ with join points, pointcuts, advices, introduction and aspects being the essential elements. But it has a different aspect language and weaving model. Also significant differences are there regarding granularity and implementation of join points.

- AspectWerkz does not extend Java to define a new aspect language. Instead it relies on its own set of class libraries, and combination of Java, XML and javadoc tags as an aspect language.

- AspectWerkz has two models for weaving namely *offline* and *online* weaving.

  - In offline weaving model, first the source code for the component code and the advices (part of the aspect language) is compiled using `javac`. Then `AspectWerkzC`, an offline weaving application provided by AspectWerkz, is used to weave the `.class` files with `aspectwerkz.xml` (a XML definition file with rest of the aspect declarations). This gives the woven classes that can be loaded and executed in a Java Virtual Machine.

  - In online weaving, `ProcessStarter` - an application provided by AspectWerkz takes the component code, code for the advices and `aspectwerkz.xml` and weaves them online and gives executable bytecodes as results without producing intermediate woven class files. Online weaving makes it possible to add or remove an Advice or change the order of an Advice on a specific pointcut dynamically. Furthermore, an introduced implementation can be replaced at runtime. For all these runtime changes the target classes do not have to be reloaded or recompiled. This is something not possible with AspectJ.

- AspectWerkz uses it own classloader to load bytecodes so that it can apply aspects into the plain java classes.

Examples of aspect codes in AspectWerkz can be found in [WEBh].

3.4.3 Hyper/J

Hyper/J [OT00] does not exactly follows the basic AOP constructs but the underlying concepts are closely related to AOP concepts. Hyper/J is an implementation of the *Hyperspaces* concept that follows *multidimensional separation of concerns*. It is based on the foundations of subject oriented programming that predates AOP.

Hyperspaces approach tries to resolve the problems in established techniques for separation of concerns, that it claims, suffer from the *tyranny of dominant decomposition*. The process of multidimensional separation of concerns in Hyperspaces constitutes the following steps -

1. Explicit **identification** of any concerns of importance.
2. **Encapsulation** of those concerns.
3. Identification and management of **relationships** among those concerns.
4. **Integration** of concerns.

Hyper/J is an implementation of the Hyperspaces concept in Java and supports all the four phases mentioned above allowing multidimensional separation of concerns in Java programs.

Differences Hyper/J is much more different than Aspect/J compared to JBoss AOP and AspectWerkz. Major differences can be summarized as follows -

- AspectJ is a Java language extension while Hyper/J works on standard Java.
- Hyper/J does not make a separation between aspect code and component code. In Hyper/J any two pieces of code can be combined together.

- Hyper/J has a different development process that goes as follows -

  1. First a developer writes the *project specification* that defines the *hyperspace* for the project that specifies the classes to be included in the hyperspace.
  2. Second, *concern mappings* are defined where each *concern* allows the developer to define a new *dimension of separation* besides separation along the *class dimension*. A class dimension is automatically defined once classes are included in the hyperspace.
  3. The final step is to specify the *hypermodules* as a collection of *hyperslices*. Hyperslices encapsulate the set of *units* belonging to the concern mappings and makes them *declaratively complete*. A unit can be any syntactic construct in Java, for example, a method. The concern mappings are identified in the *hyperspace* or the *concern matrix* as specified in the step 2 above. Finally, the *integration relationships* (composition rules) among the *hyperslices* and their *units* glue them together in a *hypermodule*.

- Hyper/J works on bytecodes and does not has a distinct weaving model. Once hypermodules are defined, executable classes can be generated out of the hypermodules using the composition tool in Hyper/J.

An example in [OT00] demonstrates how *program features* can be defined as one of the dimensions to be considered for separation. This enables the developer to use Hyper/J to come up with a decent product line system for the software where s/he can compose different versions of software with different program features as those features have been cleanly separated as distinct concerns in appropriate concern mappings.

3.4.4 Other Implementations

Currently there are many other AOP implementations besides those discussed earlier and in several other languages besides Java. Few other AOP implementations in Java are Handi-Wrap, JAC and Jiazzi. Implementations in C, C++, Ruby, Smalltalk, Delphi are available too.

3.5 Comparative Summary

AspectJ till now has the most extensive aspect language and supports weaving on source codes as well as on compiled classes. This makes the framework powerful and heavy weight on the other hand. It is too generic for any domain specific implementations but possibilities might exist for such attempts.

JBoss AOP is criticized to be lacking a fine grained join point model as is quoted in one of the web logs [WEBi] I came across

> "..They [5] are simply advising *everything* and are then (at runtime) making the decisions if an advice should be invoked or not. This is really amazing, since the key to a good AOP framework lies in how expressive and rich the join point model is. Those that have used AOP for some time will agree on that it is not enough to pick out join points on class level only:.."

There is a major difference between the motivation behind JBoss and AspectJ AOP frameworks. AspectJ from its origin carries the legacy of compiled weaving process that says aspect code should be compiled into some native form and rely less on things like reflection [Lop02]. JBoss on the other hand was designed with runtime and dynamic weaving as a primary objective [WEBj], thats why it relies upon the meta-level programming facility for Java given by Javassist.

[5]JBoss people

[Tu] is a report (not published officialy) that compares AspectJ and AspeckWerkz to some moderate detail. Here is the author's conclusion from the report -

> " Generally speaking, AspectJ is better than AspectWerkz in expressivity, usability, modularity, and performance. AspectJ is easier to share codes among advice in one aspect than AspectWerkz.
>
> If you need to create, add, remove, rearrange Advice, or replace an introduced implementation at runtime, AspectWerkz is a good choice. AspectWerkz has online weaving mode that AspectJ has not. AspectWerkz can associate advice and pointcuts through both the primary code and the aspact code, whereas AspectJ can associate advice and pointcuts only through the aspect code. AspectWerkz is also easier to share advice among aspects than AspectJ. "

Hyper/J (or Hyperspace) still seems to be a research prototype. It primarily focuses on software evolution and software product lines. Though it is supposed to address a broader range of goals regarding separation of concerns claiming *aspects* to be one of the *dimensions* it covers, the work being done seems to be too little in this area compared to the ambitious goals it have.

In general there seems to be two major ways in which an AOP framework can differ from another. First, it can be different in terms of the Aspect Language, that can be as complex as AspectJ or simple representation in XML as in JBoss AOP or AspectWerkz. Second, it can be different regarding the weaving model it has. It can either be static (compile time) weaving or dynamic (loadtime/runtime) weaving.

4 In Retrospect

4.1 Following From The past

Techniques of historical significance to AOP described in section 2.1 are not defunct. Works on these areas are being continued and some of the recent efforts demonstrate that these pre-AOP concepts are equally capable to realize core AOP constructs as established by AspectJ.

[KL01] discusses how AOP constructs can be implemented using Adaptive Programming. Similarly [BA99] and [AT98] discuss AOP issues with composition filters. Meta object protocols still directly influence most of the AOP frameworks (like JBoss AOP and AspectWerkz) as they depend on reflection techniques to achieve run time weaving supported by some kind of meta object protocol.

4.2 What Left Behind

AOP has been significantly popular in the object oriented community and is staring to get a lot of attention in the software community. AspectJ is the most extensive generic framework for AOP so far. However, the principal idea conceived by the inventors of AOP [KLM+97] does not seem to be exactly reflected in recent AOP implementations. The original definition seems to be taken over by the dominant practice in the industry as these new implementations do not particularly focus on domain specific code enhancements as exemplified in [KLM+97]. Code efficiency and domain specific aspect languages are two of the original AOP issues that seem to be vanishing away.

5 The Future Of AOP - Research Directions

AOP definitely has gained popularity in academic research and industrial applications. There are several conferences and workshops solely dedicated to AOP. [WEBk] lists a handful of references and internet links for all kind of research activities going on in AOP.

AOP has been patented by Xerox [WEBl] and it is yet to see what effect this will have on the growth of AOP as such. Nonetheless current research community is flooded with research opportunities in areas pertaining to AOP.

Some of the major research issues in the AOP community are listed below.

5.1 AOP And The Software Life Cycle

Till now AOP has been a programming issue or, more or less is practiced at the implementation phase of the software life cycle. However, there has been publications and research efforts to link and apply AOP with other phases of software development. Aspect oriented architectures, AOP with design patterns, extending modeling languages like UML for AOP, developing formalisms and metrics for AOP are few examples in this category.

These directions might help to discover a link between higher level phases like architectural design and AOP so that the crosscutting issues can be separated and visualized at higher levels in software and then successively be mapped to lower levels, starting from cross cutting issues in scenarios during requirements analysis to design artifacts to the source code.

5.2 Domain Specific AOP

Some of the researchers claim that as general purpose AOP is at rise with frameworks like AspectJ, the trend is making Aspect languages more low-level and is making the aspect languages lose the finer granularity of join points. [DVBMD] is one of the paper raising such issue and proposes the use of Logic Meta Programming for domain specific AOP.

Efforts to build AOP toolkits or architectures that can help developers to come up with (or extend) a join point model or an aspect language (in a very short time) that better suits a specific domain might result in achieving goals like code efficiency and higher performance. This would be retrofitting the concerns originally addressed by AOP that seems to be blurring away as the industry is focusing more on issues like dynamic or runtime weaving models with general purpose AOP frameworks.

5.3 Fluid AOP

[Kic] introduces the notion of *Fluid AOP* in contrast to the *Static AOP* that AspectJ provides. Following goals are listed for Fluid AOP in [Kic] -

- To be able to dynamically reform a program into a crosscutting view structure.
- Quickly prototype tools that temporarily localize the non-modular implementation of a crosscutting concern.
- To make it possible to shift back and forth between the scattered view, the temporarily localized view, or the AspectJ style modular implementation.

  With features like these, having multidimensional perspectives for program constructs and cross cutting issues would be one of the possibilities, yielding a better understanding and visualization of programs.

5.4 Beyond AOP

[LDLL03] presents another interesting vision to extend AOP to *naturalistic programming* that will possibly take the expressiveness of programming languages to the next level. It presents arguments for using ideas from linguistics like *anaphoric relations* and *binding theory* to strengthen the expressive capability of high level programming languages in terms of *temporal referencing*.

5.5 Challenges

[TDBL] gives thorough discussions on some challenging issues that were realized in the *Workshop on Aspects and Dimensions of Concern* held at ECOOP [6] 2000. Some of major the topics for discussions were as follows -

- Semantically correct composition of aspects and concerns.
- Generic representation of aspects as first-class reusable entities.
- Issues in configuration and instantiation of aspects and concerns.
- Syntactic and semantic issues for join point specification and context-sensitive join point selection.
- Need for dynamic join points for non-invasive addition of new concerns or aspects to an existing software system and ease of evolution in the presence of advanced separation of concerns.
- Need for context-sensitive join points in exception handling and systems with timing constraints.
- The need for multiple levels of abstraction and stepwise refinement in advanced separation of concerns control.

[6]European Conference for Object-Oriented Programming

A thorough investigation to find recent developments in AOP that resolves these issues and put forwards new challenges can be a research topic for yet another survey. Meanwhile, a cursory overview of available features in current AOP frameworks does imply that some of the issues are being taken care of. The dynamic join point model of AspectJ and its aspect language is evolving to address the semantic and syntactic concerns for join points. Use of meta-level programming techniques in JBoss and AspectWerkz seem to be addressing issues like non-invasive addition of new concerns.

6 Conclusions

Looking into the trends that have been followed in the programming community it can be said that the techniques for *Separation of Concerns* are now trying to evolve out of the boundary of classical abstraction and one dimensional view for separation of concerns. Much of the advances in this regard are being made at the implementation or language level.

Observing the techniques that were referred as the mechanisms for separation of concerns in most of the literature, following proposition can be set forth -

> " Current trends for Separation of Concerns employ techniques to enhance the capability of a programming language to express temporal references between program constructs during execution so that these references can be captured in a separate logical unit(s), that allows the programmer to represent the flow of program that cross-cuts multiple program components and come up with a separate representation of such cross-cutting aspects. Organization and manipulation of such aspects with base program code should result in a more understandable code. When transformed and executed as a program the result should be optimized and efficient execution."

I believe the current trend for advanced separation of concerns will give birth to more expressive languages to write programs and eventually will result in more powerful methodologies and architectural support for developing software with such languages.

The comparison of current frameworks in this paper is limited to a cursory comparison of features available among those frameworks. With a more experimental setup and good case studies much more can be revealed about the pros and cons of different styles of implementing an AOP framework. This paper however traces the way AOP has moved out from research into practice and also touches upon some of the more interesting research arenas that are being created in the field.

In conclusion, what seems more important is the way AOP has created communities among the software and language groups that are trying to take the research ahead in new directions. A balance and coordination among such research efforts inevitably will take the current programming and software practice to the next level.

References

[AT98] M. Akşit and B. Tekinerdoğan. Aspect-oriented programming using composition filters. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology, ECOOP'98 Workshop Reader*, page 435. Springer Verlag, July 1998.

[AWB$^+$93] M. Akşit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Processing*, pages 152–184. Springer-Verlag Lecture Notes in Computer Science, 1993.

[BA99] Lodewijk M. J. Bergmans and Mehmet Akşit. Analyzing multi-dimensional programming in AOP and composition filters. In *Workshop on Multi-*

Dimensional Separation of Concerns (OOPSLA 1999), November 1999.

[BB03] Bill Burke and Adrian Brock. Aspectoriented programming and jboss. Published on The O'Reilly Network (http://www.oreillynet.com/) http://www.oreillynet.com/pub/a/onjava/2003/05/28/aop_jboss.html, May 2003.

[Ber94] L. Bergmans. The composition filters object model. Technical report, Dept. of Computer Science, University of Twente, 1994.

[Dij76] Edsger W. Dijkstra. *A discipline of programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[DVBMD] K. De Volder, J. Brichau, K. Mens, and T. D'Hondt. Logic meta-programming, a framework for domain-specific aspect programming languages. http://www.cs.ubc.ca/k̃dvolder/binaries/cacm-aop-paper.pdf.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GKB91] Jim des Rivieres Gregor Kiczales and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991. 335 pages.

[HL95] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming—a critique of pure objects. In *Proc. 1993 Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, September 1993.

[Ker] Mik Kersten. Ao tools: State of the (aspectj) art and open problems. http://www.cs.ubc.ca/ murphy/OOPSLA02-Tools-for-AOSD/position-papers/kersten.pdf. Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, Mik.Kersten@parc.com.

[Kic] Gregor Kiczales. Aspect-oriented programming the fun has just begun.

[Kic92] Gregor Kiczales. Towards a new model of abstraction in the engineering of software. *International Workshop on Reflection and Meta-Level Architecture, Tama-City, Tokyo, Japan*, November 1992.

[KL01] Johan Ovlinger Karl Lieberherr, Doug Orleans. Aspect-oriented programming with adpative methods. *Communication of the ACM*, 44(10):39 – 41, October 2001.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[LDLL03] Cristina Videira Lopes, Paul Dourish, David H. Lorenz, and Karl Lieberherr. Beyond aop: toward naturalistic programming. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 198–207. ACM Press, 2003.

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[LK98] Crista Videira Lopes and Gregor Kiczales. Recent developments in AspectJ. In *Workshop on Aspect Oriented Programming (ECOOP 1998)*, June 1998.

[LL94] Cristina Videira Lopes and Karl J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In M. Tokoro and R. Pareschi, editors, *Proc. 8th European Conf. Object-Oriented Programming*, pages 81–99. Springer Verlag LNCS 821, July 1994.

[Lop97] Cristina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[Lop02] Cristina Videira Lopes. Aspect oriented programming: A historical perspective. ISR Technical Report UCI-ISR-02-5, University of California, Irvine, Institute of Software Research, 2002.

[OT00] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proc. Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, December 1972.

[TDBL] Peri Tarr, Maja D'Hondt, Lodewijk Bergmans, and Cristina Videira Lopes. Workshop on aspects and dimensions of concern: Requirements on, and challenge problems for, advanced separation of concerns. *Lecture Notes in Computer Science*, 1964.

[Tu] Yan Tu. Comparison between aspectj and aspeckwerkz.

[WEBa] Eclipse project site (http://www.eclipse.org).

[WEBb] AspectJ official site (http://www.aspectj.org).

[WEBc] The AspectJ Programming Guide (http://www.aspectj.org).

[WEBd] The AspectJ Development Environment Guide (http://dev.eclipse.org/viewcvs/indextech.cgi/c̃heckout/̃aspectj-home/doc/devguide/index.html).

[WEBe] Online Documentation for JBoss AOP (http://www.jboss.org) http://www.jboss.org/index.html?module=html&op=userdisplay &id=developers/projects/jboss/aop#framework.

[WEBf] JBoss Java Application Server site (http://www.JBoss.org).

[WEBg] Javassist home page (http://www.csg.is.titech.ac.jp/c̃hiba/javassist/).

[WEBh] AspectWerkz web site (http://aspectwerkz.codehaus.org/).

[WEBi] Weblog: JBoss AOP is lacking a decent join point model (http://blogs.codehaus.org/people/jboner/archives/000063.html).

[WEBj] Interview with Bill Burke on AOP and JBoss (http://www.theserverside.com/events/library.jsp#burke).

[WEBk] Aspect Oriented Software Development - official site (http://www.aosd.net/).

[WEBl] United States Patent 6,467,086 (http://patft.uspto.gov/).