

.A.O.P.

The Next Generation Of

.O.O.P.

By Sandeep Zechariah George K

And

Sanyo John K

Index

Introduction - 1

List Of Paradigms - 1

Relation to OOP & Other Paradigms - 2

Separating Aspects & Components - 3

What Is AOP ? - 4

The Problem & The Solution - 5

Workflow Adaptation Using AOP - 5

The ASPECTj Language - 8

What Are Aspects ? - 10

Basic Functionality - 10

Component Language & Program - 11

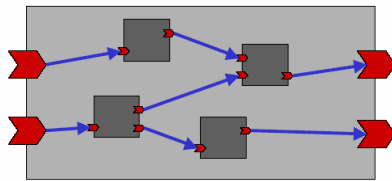
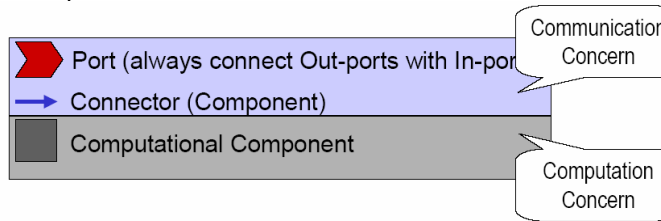
Summary - 12

Conclusion - 12

References - **13**

INTRODUCTION:

Aspect-Oriented Software Development is a new approach to software development that addresses limitations in Object-Oriented Software



Development, namely the loss of system modularity in the face of "concerns" that cut across the boundaries of the main object model decomposition, which is the dominant form of modularity used in OO applications. For example, security is a common concern that requires consistent handling across a collaborating set of objects, in some, but not

necessarily all circumstances. AOSD offers a new form of modularity designed to encapsulate these cross-cutting concerns and thereby restore overall system modularity. These concerns or *aspects* are identified, modularized, developed independently, and then combined with the main object model in a structured way to compose the application, a process called *weaving*. Aspect-Oriented Programming (AOP) will revolutionize how software is developed in the coming years. If you haven't heard about AOP or don't know what it is about, then prepare for inspiration: AOP is a simple and elegant construct with the capability of really changing the way you develop software. As a programming construct like inheritance or recursion, the language in which you develop applications must support it for you to be able to harness its capabilities. As you have probably gathered from the title of this article, C# supports AOP, as does Java. Few other languages have built-in support for AOP, so if you aren't coding in either of these languages then you might just find yourself wanting to learn one after you read this!

A **programming paradigm** is a paradigmatic style of programming (compare with a Methodology which is a paradigmatic style of doing software engineering).

The programming paradigm involved provides (and determines) the view that the programmer has of the execution of the program: in the case of object-oriented programming, for instance, the programmer sees the execution of the program as a collection of dialoguing objects, while under functional programming the execution is seen as a sequence of state-less function evaluations.

Just as different schools of software engineering advocate different *methodologies*, different programming languages advocate different *programming paradigms*. Most languages are designed to support a particular paradigm (Smalltalk and Java support object-oriented programming while Haskell and Scheme support functional programming, for example), but some are capable of supporting multiple paradigms (such as Common Lisp, Python, and Oz.) The relationship between paradigms and languages can be quite complex, however: C++, for instance, adds aspects of object-oriented programming to C, a structured programming language.

Cutting and pasting code is not a programming paradigm: rather, it is a source code *editing* technique. The end result of a cut and paste session can be a program that fits any of the existing paradigms.

LIST OF PARADIGMS

Contrasted paradigms:

- Structured programming
- Unstructured programming
- Imperative programming
- Declarative programming
- Procedural programming
- Functional programming
- Value-level programming
- Function-level programming
- Flow-driven programming
- Event-driven programming
- Scalar programming
- Array programming
- Within Object-oriented programming
 - Class-based programming and Prototype-based programming
- Within Logic programming
 - Rule-based programming and Constraint programming

Non-contrasted paradigms:

- Component-oriented programming
- Aspect-oriented programming
- Relational programming
- Symbolic programming
- Table-Oriented Programming

RELATION TO OOP AND OTHER PARADIGMS

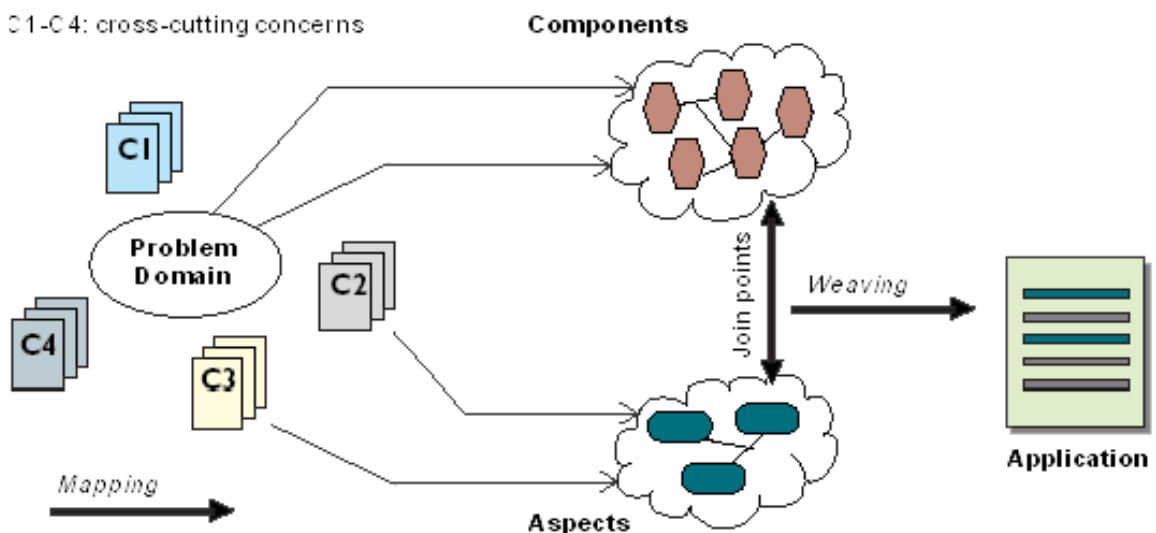
All major programming paradigms allow design and code to be structured into units of function or behavior which can be put together to produce a complete software system. These units are usually called components. Ideally, each property of the system is encapsulated in a single component to get a clear separation of concerns. Separation of concerns reduces the complexity of software development and maintenance.

Today, one of the dominant programming paradigms is OOP, which abstracts both data and behavior into a single conceptual unit, called object. In contrast to other programming paradigms, like for example procedural programming, where each procedure usually defines a smaller task to be accomplished, the designer does not subdivide a problem based on functionality, but will look for objects used in the system. The object-oriented approach has several advantages compared to other programming paradigms, like the object-oriented analysis and design methodology, inheritance and dynamic binding that allows for polymorphism, and increased potential for reuse. Although OOP makes life easier, it does not solve all problems. Many requirements of real software systems do not decompose neatly into a single object or component. For example logging, error handling, or synchronization policies may affect the implementation of a number of

methods in a number of classes. Extensive logging will usually not be centered around a single feature of the program. For robust programs, error handling is of importance throughout the code.

Synchronization policies affect all objects that, for example, work on shared data. All of these requirements are examples of properties that are not the primary task of the affected components. These kind of cross-cutting concerns are called aspects. Because the implementation of aspects is scattered throughout the code, they may lead to complex systems that are difficult to develop and hard to maintain. .(cont....)

Figure 2.1 illustrates cross-cutting concerns.



programming paradigm like OOP. The code of several components can be seen in the columns. In the components the cross-cutting code of an aspect is highlighted. At the right is shown how AOP addresses this problem by abstracting from cross-cutting concerns by means of aspect modules. The components are still in place, but the cross-cutting code has been extracted and isolated in a single aspect module.

Post-object programming mechanisms try to find a solution for this problem. Several approaches have been proposed to tackle cross-cutting concerns. The one treated in this paper is aspect-oriented programming; AOP is a means to cleanly separate aspects and components. It does for cross-cutting concerns what OOP has done for object encapsulation and inheritance – it provides a language mechanism that explicitly captures cross-cutting structure. This makes it possible to program cross-cutting concerns in a modular way, and achieve the usual benefits of improved modularity: “it allows simpler code that is easier to develop and maintain, and that has greater potential for reuse .

Just as OOP builds on ideas previously used in other programming paradigms, like procedural programming, AOP does not reject existing technology but extends it with possibilities to capture cross-cutting concerns in a modular way. Because cross-cutting properties can exist in any software system, aspect-orientation is not limited to the object-oriented paradigm, but is also applicable to, for example,

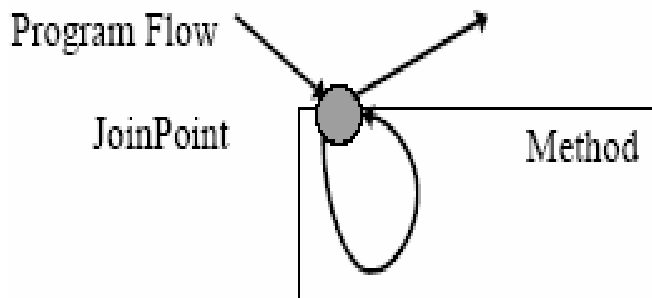
the procedural programming paradigm. An example aspect oriented language for a procedural language is AspectC , which extends C. An example aspect-oriented language for an object-oriented language is AspectJ , which is an extension to Java..

SEPARATING ASPECTS AND COMPONENTS:

Before the describing AOP in more detail, a definition of components and aspects will be given. A property is a *component* if it can be cleanly encapsulated in a unit of function or behavior, like objects, methods, procedures, or APIs. Examples of components are GUI elements, databases, and readers and writers. *Aspects*, in contrast, tend not to be units of functional decomposition, but are properties that affect the performance or semantics of the components in *systemic* ways. Typical examples of aspects are logging, error handling, and synchronization policies. It is now possible to clearly state the goal of AOP: to support the programmer in cleanly separating components and aspects from each other, by providing mechanisms that allow to abstract and compose them to produce the overall system . The mechanisms for abstraction and composition are respectively the *join point model* and *weaving*.

The join point model is a frame of reference that allows to define the structure of crosscutting concerns in aspect modules. Join points are those points in the execution of a program's components where aspects apply, thus where components and aspects are joined. The model consists of a means to identify the join points and a way to describe how aspects affect the implementation at join points.. The lifetime of a method call join point is the entire time from when the call begins to when it returns. The behavior at the method call join point can be defined just before or just after execution of the method.

Figure: Visualization of a join point in the call graph of a component. The invocation of a method is a point, called join point, in the control flow of the program where aspects can apply. The behavior at this join point can be defined just before or just after the execution of the method.



Because the concepts of the join point model are not known in traditional programming languages, a special aspect description language is needed to identify the join points and describe the behavior at those points. An example of an aspect description language for Java is AspectJ.. Other types of join points are known as well. Examples of join points that can be defined in AspectJ include method call, method execution, field accesses, object construction, and exception catching join points.

WHAT IS AOP?

AOP is a way of executing arbitrary code orthogonal to a module's primary purpose, with the intention of improving the encapsulation and reuse of the target module and the arbitrarily invoked code. AOP is best demonstrated by example, the classic one being event logging. The following example has been synthesized from Juval Lowy's article cited in the Bibliography below. Suppose you have a class Foo and you wish to write to a log file each time a particular method is called for rudimentary performance statistics or auditing purposes. You might ordinarily write code such as the following to satisfy this requirement:

```
public class Foo {
protected EventLog eventLog;
public Foo() {
// create an event log
eventLog = new EventLog();
// Name a Source
eventLog.Source = "Foo Application";
}
public void bar() {
eventLog.WriteEntry("Bar method begin");
// do bar()
eventLog.WriteEntry("Bar method end");
}
```

THE PROBLEM AND THE SOLUTION:

AOP is the solution, but the solution to what problem? Object-Oriented (OO) divide the world in different Objects and components and that can be a problem when it comes to functionality that cross cut the object world. It can be hard to modularize this into classes. Example of such functionality can be synchronization, performance optimization, exception handling etc. Another problem with OO is that you need to decide all the interfaces before any implementation can start. This is because it is hard to change an interface afterward because it may change a lot of classes. AOP solves this so you can modify the static structure of them afterward without changing any code within the classes.

WORKFLOW ADAPTATION USING ASPECT-ORIENTED PROGRAMMING:

CONTROL FLOW PERSPECTIVE

First of all we consider the control perspective and describe how the control flow can flexibly be changed using AOP. Figure 2 depicts the insertion of activities and control flow constructs, defined by the Workflow Management Coalition, such as sequence, split, join and iteration. Activity diagrams on the left side show a process fragment, while sequence diagrams to the right represent interactions between objects. Replaced control flows are depicted by dashed arrows in activity diagrams, while interactions contain some special constructs showing interceptions by the aspects. In Figure 2a an activity (A1) which is an instance of WfActivityA is replaced by the instance of WfActivityB (A2). We assume WfActivityA and WfActivityB to be subtypes of the OMG-interface WfActivity. In this case we use an aspect that has a pointcut activated by the invocation of an WfActivityA constructor performed by WfProcess instance P. The corresponding pointcut method is executed *instead* of the original invocation. In the sequence diagram the original call is depicted as a dashed connector with a transparent dot at the beginning and transparent arrow at the end. Although this call is not executed it should especially be depicted for instead-invocations to clarify what pointcut was activated. The aspectual invocation is depicted by the connector between the caller object and the aspect instance with the black dot on the aspect side. It is labelled with the original method call. In this case the pointcut-method creates an instance of WfActivityB and returns it to P instead of a WfActivityA-object (we assume the process is handling its activities through the WfActivity interface).

The context used for the creation of A2 can differ from the original data. A2 considers P as its owner process and reports it the execution results. Deletion of activities can be realized analogously by replacement by dummy activities. In Figure 2b a new activity (A2) is inserted between two existing ones and all of them are executed in a sequence. The activity constructor invocation is once again intercepted by the aspect. But in contrast to the first example the pointcut method is executed *before* the constructor. It creates a new instance of WfActivityB. This activity cannot report its results to the process, because P is not aware of its existence and it would interpret the call as the result of A1. So the result is reported to the aspect and thereafter the intercepted constructor call is executed. The context passed over to A1 can be derived from the A2 result which in that way can influence the rest of the process. Figure 2c shows an inserted AND-Split between the activities A0 and A1. A single thread of control now splits into two concurrently executing threads [14]: the old (starting with A1) and the new one (A2). In contrast to the previous case the aspect does not wait until A2 is finished before it continues the instantiation of A1. Therefore the context of A1 cannot be affected by A2, whose returning result is omitted since it is not relevant. An OR-Split (i.e. branching into several alternative threads) can be inserted analogously to the activity replacement with the help of an instead pointcut method.

FIG:IDEAL MODULARITY..

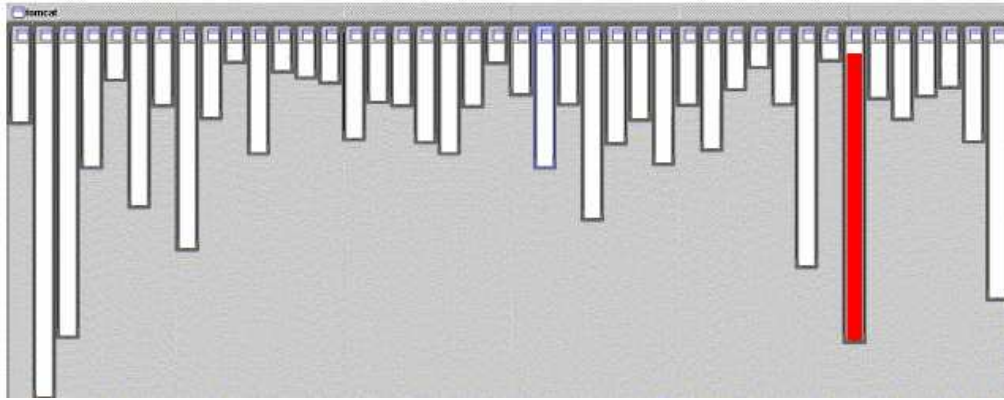
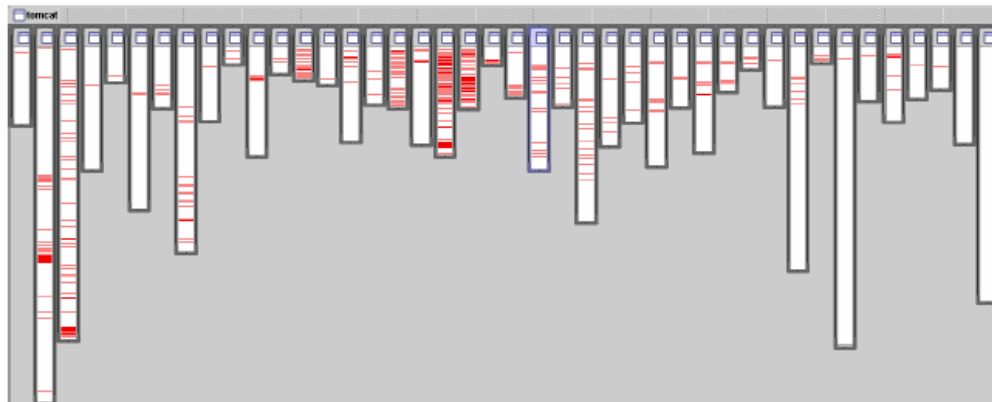
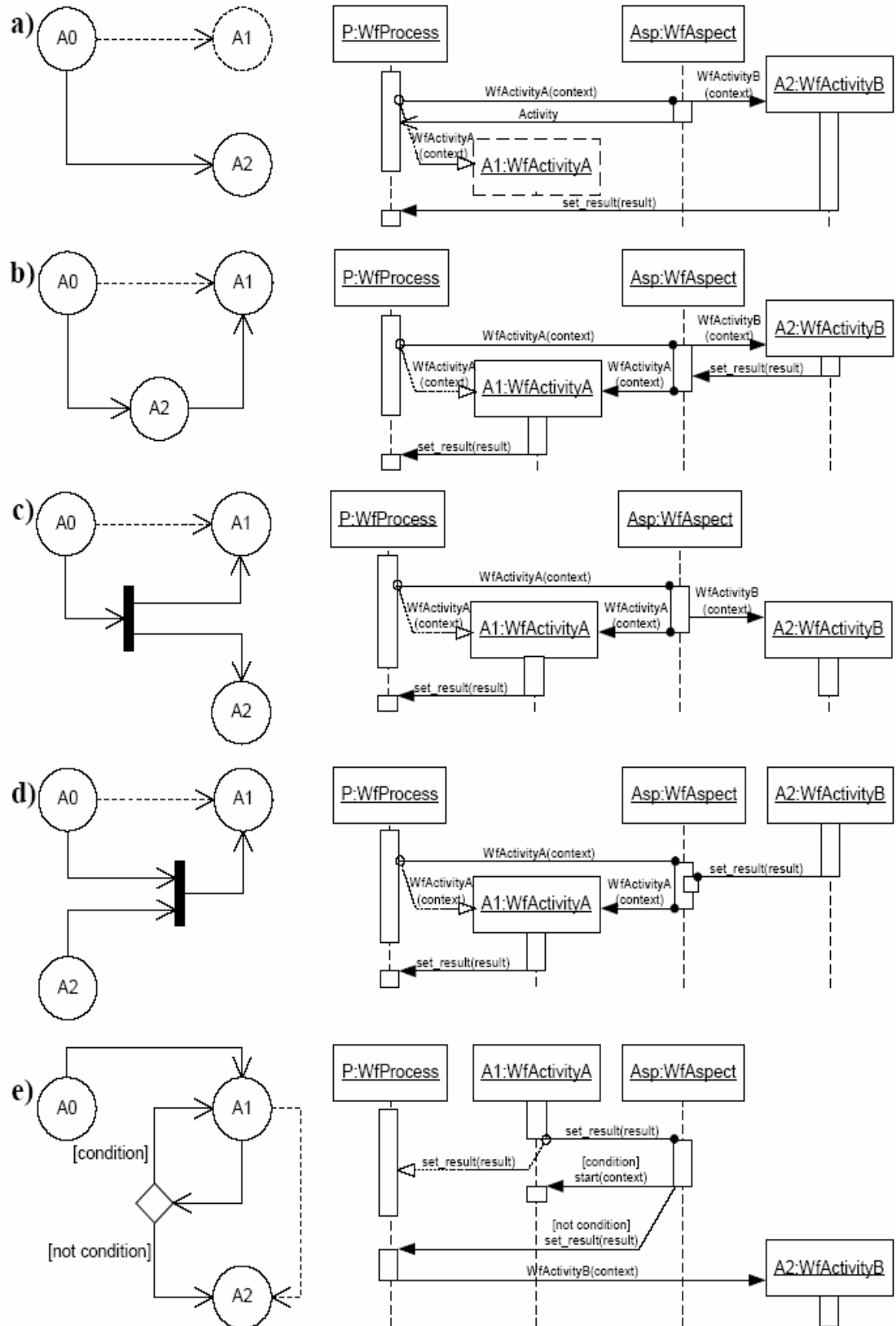


FIG:DISTRIBUTION OF USEFUL CODE AMONG MODULES WHILE LOGGING PROCEDURE IS DONE..





The method evaluates given conditions and decides on creating either A1 or A2 (XOR-Split) or both of them. Multiple threads converge into a single one by using the join construct [14]. The insertion of an AND-join that merges parallel threads is depicted in the Figure 2d. Since the coordination of A0 and A1 is already handled by the process, the aspect has to ensure that A1 can only start after A2 is finished. Therefore one pointcut observes the final call of A2 that return the result and sets an internal flag as soon as it was executed. Another pointcut

method intercepts the instantiation of A1 and lets it proceed only after the flag was set. If the converging branches are alternatives (ORJoin) the aspect has to detect the termination of the both, A0 and A2. It has to trigger the creation of A1 at the moment the first of these events occurs and has to prevent the instantiation when the second one takes place. An iteration (i.e. repetitive execution of a process segment) is added in Figure 2e. Activity A1 is performed repeatedly as long as a certain condition is fulfilled. A before pointcut detecting the result delivery of A1 and starting it again and again, is A2...

THE ASPECTJ LANGUAGE

The language described here is AspectJ. That is an implementation of AOP in Java which is developed by Xerox Parc and version 1.0 was released in November 2001. It works as a precompiler and you can see the generated code which can help a lot during development before you getting used to the new syntax. It also gives the advantage that the end users don't need to install anything special to run the programs except the virtual machine. AOP is constructed from OO and only extends the benefits that you already have without taking anything away. Your aspects can have the same functionality as you classes with instance and static methods, attributes and so on. The aspects can add functionality to the classes and/or change it without changing the code in them in any way. Where the aspect is used much depend one what it is used for. If it is temporary, part of the system or the system can function without it but better with it. One common implementation is that functionality appears in multiple methods. It may also be temporary used only during the development. Example of this is logging and error handling. The solution is to place it just on one place and use that in the entire system. This makes the code cleaner and less coding is needed. Example of functionality that can be separated into an aspect is synchronization and performance optimization. This is functionality that is often used in classes but not a part of the normal business logic. It is also aspects that can be reused in many classes and also be reused in different projects.

ASPECTS

Aspects are for AOP what classes are for OO. It gathers all the functionality inside of it. It can extend other aspects or classes in the same way as with classes.

```
Aspect ExampleAspect {  
}
```

This example doesn't do much; just declare a new aspect with the name ExampleAspect,,.

JOIN POINTS

It is with join points you decide where the aspects is executed. AspectJ includes 11 different join points. From a normal method call to more advanced join points such as "within,, and "target,,. The join points can be combined as a boolean expression. Many of the different join points take the name of the class and/or method as a argument and AspectJ accept "*",, and ".,,, as wildcards their.

```
Aspect ExampleAspect {  
  pointcut p() : call(int ExampleClass.*(..));  
}
```

A join point has been added to the previous example. This join point is fired every time a method in the "ExampleClass,, is called that takes zero or more arguments and returns an integer.

DESIGN IMPROVEMENTS

The main design improvement you get with AOP is better modularization. Redundant code can be placed in an aspect instead of copied to all classes that need it. You can concentrate on putting the business logic in the classes and the rest can be handled with Aspect. This makes the code easier to read and inspect. But the aspect can also very easy make the code harder to follow because if you don't know about the pointcuts will you don't know what's happening by just follow the logic path of the code. This can specially be a problem if the design is changed later during the development and functionality is added with aspects.

DEVELOPMENT

On big advantage with AOP is that you can put the debugging code outside the normal code and can easily turn it on when you need it and off when you compile the final version. In normal case is it hard to handle this type of code. You don't want it in the final version but you need it for maintenance so it may not be the best idea to just delete it either.

LOGGING

Put a point cut on all method calls except the output method. Then make an advice that implement *before()* and output the method name and all the parameters.

TRACING

Extend the logging functionality so it has an *after()* advice too. And add a stack that push when *before()* is called and pop when *after()* is called. With this small implementation can all method calls be tracked and faults can be found without changing the code in any ways.

PROFILING

If performance is a problem must the bottlenecks be found and this can be a time consuming task in a normal project. But with some help from AOP can this be done with a few lines of code. You only have to change the tracing example so it has a timer and outputs the difference from the *before()* to *after()* calls.

ERRORS AND WARNINGS DURING COMPILING

The AspectJ compiler has a function to find warnings and errors during compiling with the help of pointcuts. The only limitation is that it does not accept pointcuts that needs runtime information. This can be used to check that code standards is followed and classes that needs to be executed in a special way is that.

```
declare warning: call(ExampleClass.new(..))
"Constructor Called!!;
```

This example outputs a warning if the constructor of the "ExampleClass,, is called.

PERFORMANCE

Aspects can also be used to improve the performance of the system. This type of functionality is normally used within many classes and it can therefore be hard to find a good place to put them. Instead can you put them in an aspect and use that in the entire system. Example of this type of functionality is polling, caching, buffering etc.

EXCEPTION HANDLING

The normal case is that you want to terminate the current function and output an error message to the user. Because this is handled in almost the same way in the entire program can it be a good idea to put this in an aspect. The *handler()* pointcut picks an execution of an exception handler.

STATIC CROSSCUTTING

Aspects can in many ways drastically change how the original class works. Aspects can also change the static structure by adding fields and methods and even change the class hierarchies. This can be very powerful tool and you can change the original class beyond recognition.

declare parents: ExampleClass extends ExampleParent;

This example changes the "ExampleClass,, so it extends "ExampleParent,,. All methods must be implemented if new abstract methods are added; they can already be implemented or added by aspects.

WHAT ARE ASPECTS?

To better understand the origins of tangling problems, and how AOP works to solve them, this section is organized around a detailed example, that is based on a real application we have been working with [18, 22]. There are three implementations of the real application: easy to understand but inefficient, efficient but difficult to understand, and an AOP-based implementation that is both easy to understand and efficient. The presentation here will be based on three analogous but simplified implementations. Consider the implementation of a black-and-white image processing system, in which the desired domain model is one of images passing through a series of filters to produce some desired output. Assume that important goals for the system are that it be easy to develop and maintain, and that it make efficient use of memory. The former because of the need to quickly develop bug-free enhancements to the system. The latter because the images are large, so that in order for the system to be efficient, it must minimize both memory references and overall storage requirements.

BASIC FUNCTIONALITY

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

- 1 In some communities this term connotes the use of functional programming languages (i.e. side-effect free functions), but we do not use the term in that sense.
- 2 We have chosen Common Lisp syntax for this presentation, but this could be written

fairly easily in any other Algol-like language.

	Observer	Visitor	Decorator
Kinds of aspects	Concrete instance extends reusable abstract aspect.	Concrete instance extends reusable abstract aspect.	Decorator directly implemented by means of aspect.
Assignment of roles	Parent declaration for subject(s) and observers.	Parent declaration for elements to be visited.	-
Attached functionality	-	Introduction of accept methods into elements.	-
Event handling	Concretize an abstract pointcut to trigger events.	-	Pointcut in aspect identifies execution points of interest.
Positive points	Use of existing classes. Change dependencies.	Use of existing subject classes.	Use of existing subject classes.
Negative points	-	Only two classes supported by pattern protocol.	No dynamic composition. Less reusable.

FIRST EXAMPLE OF AOP

In this section we return to the image processing example, and use it to sketch an AOP-based re-implementation of that application. The presentation is based on a system we have developed, but is simplified somewhat. The complete system is discussed in [22]. The goal of this section is to quickly get the complete structure of an AOP-based implementation on the table, not to fully explain that structure. Section 6 will provide that explanation. The structure of the AOP-based implementation of an application is analogous to the structure of a GP-based implementation of an application. Whereas a GP-based implementation of an application consists of: (i) a language, (ii) a compiler (or interpreter) for that language, and (iii) a program written in the language that implements the application; the AOP-based implementation of an application consists of: (i.a) a *component language* with which to program the components, (i.b) one or more *aspect languages* with which to program the aspects, (ii) an *aspect weaver* for the combined languages, (iii.a) A *component program*, that implements the components using the component language, and (iii.b) one or more *aspect programs* that implement the aspects using the aspect languages. Just as with GP-based languages, AOP languages and weavers can be designed so that weaving work is delayed until runtime (RT weaving), or done at compile-time (CT weaving).

THE COMPONENT LANGUAGE & PROGRAM

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

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

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

SUMMARY

In this position paper we proposed an approach for the dynamic evolution of workflow instances by using aspects. It allows flexible process adaption and reuse of both the object-oriented process implementation and the adopting aspects. The changes can either be caused externally or triggered by the auditing component that can be realized by aspects too. In that way a cyclic workflow improvement can be realized. Unfortunately the most implementations of aspect languages only support static aspect weaving at the pre- compile time . Though if using this languages a workflow has to be restarted, in order to be changed, the aspect-based adaption still allows the reuse of both primary workflow implementation and adapting aspects. Dynamic run-time aspect assignment is desirable, in order to realize automatic improvement cycle. The approaches allowing dynamic weaving are e.g. *AOP/ST* , that makes use of reflective capabilities of Smalltalk, or *Aspect Moderator Framework* , which is implemented in Java and introduces a special design pattern for objects aspects are assigned to. General purpose aspect language *Sally* is an extension of Java realized by a pre-compiler. It also supports dynamic aspect assignment at the run-time. Other potential application areas like process error handling are to be examined in the future. An open implementation issue is the aspect realization in distributed environments, which is especially important for workflow management systems.

CONCLUSION

AOP is a very powerful tool that I think will become much bigger in the future. But for now is too much of an academic project and under development to be really useful for a bigger audience. More research and standards is needed such as design and usage patterns
AOP and AspectJ have a lot of advantages and I especially like the way it helps with the coding and debugging such as the way logging and tracing can easily be implemented.

REFERENCES

[1] Object Management Group, "The Architecture of Choice for a Changing World." Available <http://www.omg.org/mda/>.

[2] Tzilla Elrad, et al., "Special Issue on Aspect-Oriented Programming", Communications of the ACM, vol. 44, issue 10, Oct. 2001.

[3] Object Management Group. "UML Resource Page." Available <http://www.omg.org/uml/>.

[4] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, Object-Oriented Engineering, ACM Press, Addison-Wesley, Wokingham, England, 1992.

[5] Rational Software, "The Rational Unified Process." Available <http://www.rational.com/products/rup/faq.jsp>.

[6] Dean Wampler, et al., "UML/EJB Mapping Specification", Java Community Process. Available <http://www.jcp.org/en/jsr/detail?id=26>.

[7] Rational Software, "Rational XDE™: Liberated Development." Available <http://www.rational.com/xde/>.

[8] K. Czarnecki and U. W. Eisenecker, Generative Programming, Addison-Wesley, Boston, Mass., 2000.

© 2003 Dean Wampler, All Rights Reserved. 11

[9] Aspect-Oriented Software Development Steering Committee, "Aspect-Oriented Software Development." Available <http://aosd.net/>

[10] AspectJ Development Team, "AspectJ" <http://www.eclipse.org/aspectj/>

WEBSITES

- AOSD.net
- parc.com
- Wikipedia.com
- Citeseer.com
- Usenet Groups

Presented By

Sandeep Zechariah George K & Sanyo John K.
Toc-H Institute Of Science And Technology
Arakkunnam – 682313
Ernakulam District
Kerala