

A Study Of Applications Of Aspect Oriented Programming

Rupali Ahuja*, Anita Goel**

Dyal Singh College, University of Delhi.

*rupali_ahuja1@yahoo.com, ** agoel@dsc.du.ac.in

Abstract-The code to implement systemic or non-functional features of software (like logging, security, authentication etc.) is often scattered across the application reducing the cohesiveness, maintain ability and quality of software. The non functional features cut the system orthogonally and are called global or crosscutting concerns. The crosscutting concerns are hard to modularize as they affect multiple systems, functions, features, modules etc. Aspect Oriented programming (AOP) is a new programming paradigm that promises the localization and modularization of these crosscutting concerns. It provides another level of abstraction by separating the implementation of crosscutting behavior in separate modular units called "aspects". AOP is now being used to implement the crosscutting concerns like performance monitoring, synchronization, caching,, security etc. In this paper, we discuss Aspect Oriented Programming. We give a brief idea of AspectJ language (an aspect oriented extension of Java). We give an overview of applications of AOP in different areas like caching, authentication, performance optimization, performance monitoring etc. We illustrate with examples, how crosscutting concerns like tracing, logging, exception handling, authorization can be implemented using AOP.

I. INTRODUCTION

Any complex software application implements both functional requirements as well as non-functional requirements (like security, authentication etc). The functional requirements are based on the business logic of application. The code to implement functional requirements is encapsulated in well defined modules. The non-functional requirements arise from required characteristic of the software (product requirement like security, logging, event monitoring etc), the organization developing the software (organization requirements like synchronization, optimization etc.) or some external sources like performance requirements (reliability, portability etc.). The code to implement non-functional requirements spans the entire architecture of system [9]. For example, the code to implement logging concern is made subpart of all methods/functions that want to log details. This results in code scattering and code tangling. It is difficult to modularize the crosscutting concerns like caching, logging, synchronization etc.

AOP [7] is an emerging software engineering paradigm that provides a solution for abstracting crosscutting code. AOP helps overcome the problems caused by code tangling and code scattering. It focuses on the separation of the various concerns of software, at the programming language level. AOP helps the programmer in cleanly separating the core concerns and the crosscutting concerns of the system. It introduces a new modular unit called *aspect* which encapsulates the functionality of the crosscutting behavior. It addresses each concern separately with minimal coupling,

resulting in modularized implementations **Error! Reference source not found..** The modular implementation results in a system that is easier to maintain, modify, test, understand, reuse and easier to evolve. AOP enables better encapsulation [6] and promotes future interoperation.

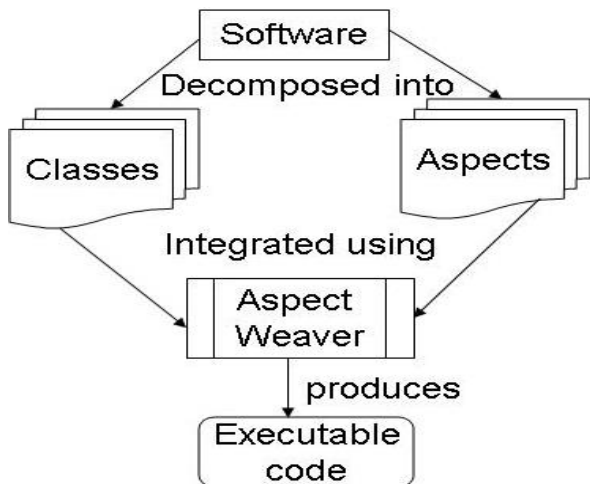
AOP is beginning to pervade all areas of software engineering. AOP is growing rapidly, and we see AOP applications in many areas such as fault tolerance, quality of service, design patterns, coordination protocols in middleware, synchronization in operating systems, performance optimization in persistent systems and databases, performance monitoring of web applications etc..

In this paper, Section 2 gives the overview of AOP. It also describes AOP terminology using AspectJ [8] language, an aspect oriented extension of Java. Section 3 discusses the different areas where AOP is effectively and efficiently being used for improving the quality of software. Section 4 states the conclusion.

II. ASPECT ORIENTED PROGRAMMING

AOP is a programming paradigm based on identification and separation of core and crosscutting concerns of software. It allows a developer to separate the code to implement specific tasks like synchronization, security etc that are present at various locations of software, from the main code. AOP introduces a modular unit called *aspect* which encapsulates the functionality of the crosscutting concerns. It provides a process and a methodological approach for defining, specifying, designing and constructing aspects. To implement programs in an aspect oriented way, we first separate the module level core concerns from the system level common crosscutting concerns. We implement each concern independent of each other. For example, we can implement business logic by developing classes using an Object Oriented (OO) language and we can implement crosscutting concerns using an aspect oriented language. The classes encapsulate the core concerns and aspects implement the crosscutting concerns. Finally, the main program code and the aspects are integrated into a final executable form using a tool called "aspect weaver" during compile/run time. Figure 1 illustrates the weaving of aspects and classes to get the executable code. Aspect Oriented Programs can be implemented using various aspect oriented tools. These tools provide a mechanism for declaring, defining, composing and abstracting aspects. They differ in the way they declare aspects. Some tools [13] are extension to programming languages. AspectJ [8] is an aspect oriented extension to Java programming language, AspectC++

is an aspect oriented extension of C++ and AspectC is an



aspect oriented extension of C.

Figure 1: Weaving classes and aspects to produce final executable code.

Some tools declare aspects using annotations or XML, like AspectWerkz supports both annotation and xml based aspect declaration and, JBOSS and spring framework use XML-based aspect declaration style. In the following subsection, we give an overview of AspectJ language.

AspectJ

AspectJ [8] is a general purpose aspect oriented extension for Java. It is an open-source project initiated by PARC and now led by IBM [2]. It can be downloaded freely [1]. It was built to provide support of AOP in java programs. AspectJ follows the join point model. Join points specify the points where crosscutting code is plugged in the main code. We give a brief description of various language constructs of AspectJ.

Join point

Join points represent well-defined points in the execution flow of a program. A join point defines an event in a system where a crosscutting concern can be joined with the core concerns at compile or run time. Many points in the execution flow of the program can act as join points. Some of them are Constructor/Method call, Constructor/Method execution, field set and get points, Object/Class initialization and Exception handling points. Join points may contain other join points like one method may call several other methods before it returns. For example, in a tree program, a join point can be a call or execution of methods insert, delete, find etc.

Point cut

A point cut selects a subset of join points where crosscutting code will be embedded within core code. It is a language construct that picks out a set of join points based on

a defined criterion. The criteria for selecting a join point can be specified by an expression which may include wildcards like “*”, “+”, “..” etc. and operators like “||”, “&&” etc.. For example, in Table 1 statement (1) represents the “TreeMethods()” point cut. This point cut selects the join points that represent the execution of the methods of class “tree”.

Advice

Advice is a construct indicating code that should run at each join point. Advice is a code that executes before, after, or around a join point. A “before()” advice gets executed before the code originally at the join point, and “after()” advice gets executed after the code originally at the join point. An “around()” advice surrounds the code at the join point and it is up to the programmer to decide when, if ever, to call the original code using the “proceed()” method. Advice is similar to a method of a class but contains the code of a crosscutting concern.

For example in Table 1, statement (2) represent the before advice to implement the tracing crosscutting concern. This code will be implemented before execution of methods of “tree” class.

Introduction

An introduction **Error! Reference source not found.** is used to introduce new fields, methods or constructors to a class or interface. It makes static changes to the modules that do not directly affect them. Therefore they are also referred to as inter-type members.

Introduction of field and methods can be done by defining them in an aspect and prefixing them with the class name to which they need to be introduced. For example, a new variable “nElem” can be introduced to store the number of elements in “tree” data structure and method “numElem()” can be introduced in class “tree” to find the number of elements inserted in tree.

```

aspect TreeAspect{
public static int tree.nElem=0;
public static int tree.numElem()
{return tree.nElem;}}
    
```

Aspects

Aspect is a program unit similar to the “class” of OOP but is specifically meant to implement the functionality of a required crosscutting concern. Aspects of a system are independent elements that can be changed, inserted or removed at compile time, and even reused without affecting the rest of the system. Table 1, 2, 3 and 4 display the aspects required to implement tracing, logging, authentication and exception handling concerns respectively.

III APPLICATIONS OF AOP

The applications of AOP are rapidly growing. It is being used to implement many crosscutting concerns. Here we

discuss some areas where AOP is effectively being applied to improve the quality of software.

A. Security

The security of a software system is an attribute that permeates the whole system. As such, any attempt to address security [10] concerns in a software system must, of necessity, be global in nature, and security solutions must be applied consistently at every relevant location. Using OOP, we can write a central SecurityManager class for an application. However, calls to this class and important checks will necessarily be spread throughout the code base. If a critical check is forgotten in an important spot in the code, the central class will not have an opportunity to recover. If security consideration is omitted in just one place, it can easily lead to a flaw. This problem exists because security is a concern that affects the entire system in a broad way. AOP solves this problem by allowing security concerns to be specified in a modular way and applied to the main program in a uniform way.

B. Tracing

Tracing is done to know the logical flow of the program during execution. Tracing determines the various application components that were involved in a software task. Tracing is required to troubleshoot functional and performance problems. To trace the execution flow, usually print statements are inserted in all the components of software. AOP separates the lines of code that are required for tracing the execution flow, from the rest of application. It encapsulates the tracing code in an aspect.

Table 1: A Tracing Aspect

```
import org.aspectj.lang.*;
public aspect TraceAspect{
    pointcut TreeMethods() : execution(* tree.*(..)) ---(1)
    before() : TreeMethods(){-----(2)
        Signature sig =(Signature)
        thisJoinPointStaticPart.getSignature();
        System.out.println("Entering method "+
            +sig.getName());}
```

Table 1 displays an example of tracing aspect. This aspect prints a message and the name of the method of class “tree” before it is executed. The tree software is weaved with aspect shown in Table 1 using AspectJ compiler (ajc). Figure 2 displays the output of the integrated software.

C. Logging

Logging of software is required to get the internal details of execution for the purposes of understanding the behavior of software, testing, debugging etc. For logging, there is a need to access the internal execution details during execution of the application software, record these details in a file, and, then

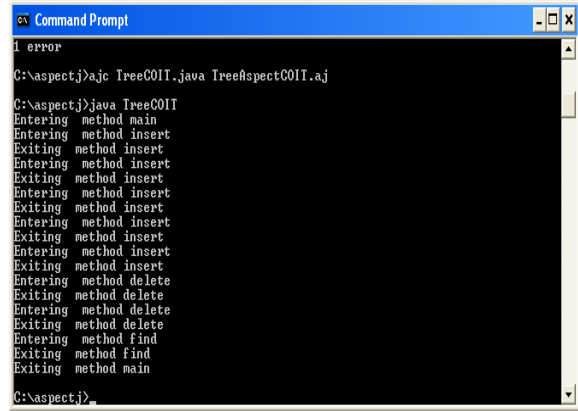


Figure 2: Output of tracing concern

view the recorded execution details **Error! Reference source not found.** Provisions are required to be made in the software to facilitate the logging activity. In order to log the software, additional statements are required to be inserted into the software to capture the internal execution details. These statements are inserted at various locations of the program. Logging is a crosscutting concern as the logging technique affects every single logged part of the system. AOP provides a non-intrusive approach to logging of software.

Table 2: A Logging Aspect

```
import org.aspectj.lang.*;
import java.util.logging.*;
public aspect LogAspect{
    private static Logger mylog
    =Logger.getLogger("Logging details");
    pointcut treeMethods() : execution(* tree.*(..));
    after() : treeMethods(){
        Signature sig=(Signature)
        thisJoinPointStaticPart.getSignature();
        Object args[]=thisJoinPoint.getArgs();
        if(args.length == 0)
            msg="Entering method " + sig.getName();
        else
            strArgs="";
        for (int i=0;i<args.length;i++)
            strArgs = strArgs + args[i].toString() + " ";
        msg="Entering method " + sig.getName() +
            "with argument " + strArgs;
        mylog.logp(Level.INFO,sig.getDeclaringType
            ().getName(),sig.getName(),msg);}
```

Table 2 displays an aspect that logs the name of method and its arguments values after it is executed.

D. Design Patterns

A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems [14]. It describes the problem, the solution, when to apply the solution, and its consequences. Many patterns are crosscutting. Patterns can affect multiple classes, they can also be invasive and hard to

(re)use. AOP localizes the code for a design pattern, makes patterns lighter, more flexible, and easier to (re)use

E. Authorization and Authentication

Authentication determines whether a user has access to the components of particular software. Authorization is a post authentication process that determines what users are allowed to do [3]. It covers what functionality is available after they have been given access to a system. The authorization policies govern groups of systems, roles, users and commands. The policies as a whole must be flexible and cohesive enough to enable corporate-wide changes to be defined and enforced with ease. The advantage of using AOP for implementing authorization is that corporate-wide authorization policies can be put in place and enforced across many applications to facilitate the corporate view being mapped onto the application system infrastructure. Further to this, once these services are in place, modification of these policies can be done at a central place with no changes to the underlying applications required.

Table 3: An Authentication Aspect

```
public aspect AuthAspect{
    pointcut auth():call(* test.*(..));
    void around():auth(){
        if(user.authenticated())
            proceed();}
}
```

Table 3 displays an authentication aspect. The aspect uses an *around* advice that calls the methods of “test” class (using *proceed()* method), after determining the authentication of its users.

F. Performance Optimization

The performance of relational database applications often suffers. The reason is that query optimizers require accurate statistics about data in the database in order to provide optimal query execution plans. The issue of updating the statistics for database optimization is a crosscutting concern. Hohenstein. O propose to use aspect-orientation to automate the calculation.

Efficient data retrieval from databases is a significant issue of the design of persistent systems. Yasuhiro Aoki **Error! Reference source not found.** has proposed an aspect oriented persistent system named AspectualStore. AspectualStore opens up its data retrieval mechanism so that developers can customize it in an aspect for performance optimization. The aspect controls the generation of SQL queries and minimizes the number of round-trips.

G. Performance monitoring

Modern applications are typically complex, multithreaded, distributed systems that use many third-party components. On such systems, it is hard to detect (let alone isolate) the root causes of performance or reliability problems, especially in production **Error! Reference source not found.** Performance monitoring solutions must work under constraints imposed by the environment. They must balance

conflicting requirements such as overhead vs. richness of information. All of these need careful understanding of both requirements and solutions as well as the costs of any tradeoff decisions. Good performance is an important requirement from the business viewpoint. Supporting this requirement needs application profiling during the development phases and performance monitoring after deploying the application. AOP lets you define point cuts that match the many join points where you want to monitor performance. One can then write advice that updates performance statistics, which can be invoked automatically whenever one enters or exits one of the join points.

H. Synchronization

Synchronization is an important aspect of concurrent object-oriented systems, but treating synchronization as a single monolithic aspect leads to inflexibility and limited possibilities for reuse. Holmes et al. [5] suggest that synchronization has a number of different aspects, and introduce the 'synchronization rings' model which allows the aspects of a synchronized object to be specified independently. Separating the different aspects of synchronization provide flexible, generic implementations of common synchronization constraints, which can be reused in different contexts

I. Exception Detection and Handling

An exception is a behavior of the system indicating that the operation in process cannot be successfully completed, but from which other parts of the system can try to recover or chose to ignore. The code for detecting and handling exceptions often tangles the main code. In their study, Martin Lippert et al [11] found that AOP supports implementations that drastically reduce the portion of the code related to exception detection and handling. They found that AspectJ provides better support for different configurations of exceptional behaviors, more tolerance for changes in the specifications of exceptional behaviors, better support for incremental development, better reuse, automatic enforcement of contracts in applications that use the framework, and cleaner program texts.

Table 4: An Aspect to handle exception

```
public aspect ExcAspect{
    after() throwing(Exception e):call(* test.*(..));
    { System.out.println("Exception " + e + " caught while
    Executing " + thisJoinPoint());}
```

Table 4 displays an aspect to handle exceptions. This aspect has an *after() throwing()* advice that prints a message after an exception is thrown in the methods of the “test” class.

J. Coordination Protocols in middleware

Coordination plays a central role in technologies used to develop distributed applications such as component technologies, web services and agent technologies. Coordination protocols are scattered through the several components participating in an interaction. Mercedes et al [12] have proposed an aspect-oriented technique to separate

coordination as an independent entity in middleware. The coordination aspect is defined as an entity that encapsulates the interaction pattern or coordination protocol that governs the communication and interchange of information among two or more software entities.

K. Caching

Object caching provides a mechanism to store frequently accessed data in memory, minimizing the calls to back-end database, and resulting in significant improvement in the application performance. It also gives us the ability to refresh different types of data at different time intervals (based on pre-defined eviction and cache refresh policies). Srinu Penchikala **Error! Reference source not found.** provide an example of object caching as an Aspect in J2EE applications and discuss the steps involved in injecting the caching functionality into a sample web application. This example shows how fast the data access can be when we use a cache to store frequently accessed objects. By making caching an Aspect we get the flexibility of dynamically adding caching ability in a J2EE application for cached objects. We can also remove caching out of the application whenever it becomes a bottleneck in terms of memory usage.

L. Quality Of Service

Quality of Service (QoS) addresses the non-functional issues of services in distributed processing systems. QoS concerns originate from distribution effects such as transmission errors, dynamic bandwidth, overload situations, partial failures, etc. Becker et al. [4] discuss QoS as an aspect of distributed programs, and show how CORBA can be extended by QoS management. They have used an aspect-oriented approach for QoS integration in CORBA. The framework developed uses an aspect-oriented view on QoS to reduce the complexity of the distributed system.

CONCLUSION

AOP is a programming paradigm that addresses the problem of modularization of crosscutting concerns of software. AOP makes it possible for developers to write modular code for concerns like synchronization, logging, security, performance monitoring etc. thereby increasing the quality of software. AOP is changing the way programs are being written. It is widely being used to handle concerns like tracing, logging, caching, authorization, design patterns, security, quality of service, performance monitoring and optimization. AOP has tremendous potential for building the software of the future.

REFERENCES

[1] AspectJ homepage, www.eclipse.org/aspectj
[2] A.Colyer, A.Clement, 2005 "Aspect Oriented Programming with AspectJ". IBM systems journal, vol. 44, No.2, pp301-308

[3] Andrew Meehan, 2003. "Authorization-An aspect Oriented Service".
http://www.forge.com.au/Research/white_papers/authorize.pdf
[4] C.Becker and K. Geihs, 1998. "Quality of service – aspects of distributed programs". In Proceedings of the Aspect Oriented Programming workshop at ICSE'98
[5] David Holmes, James Noble, John Potter, 1997 "Aspects of Synchronization", TOOLS 25, proceedings of Technology of Object-Oriented Languages and Systems, Melbourne, Australia
[6] Dharma Shukla, Simon Fell and Chris Sells, 2006. "Aspect oriented Programming enables Better Code Encapsulation and reuse",
<http://msdn.microsoft.com/msdnmag/issues/02/03/AOP/default.aspx>
[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, J. Irwin, 1997. "Aspect-Oriented Programming". In proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP), LNCS, Vol. 1241, Sp:ringer Verlag, pp 220-242, Finland, June 1997.
[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold, 2001. "An overview of AspectJ". In Proceedings European Conference on Object-Oriented Programming, volume 2072 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, and New York, pp 327–353.
[9] Jay Gattani, 2004. "An Analysis of Aspect Oriented Programming with AspectJ". Technical report, University of Houston, www2.cs.uh.edu/~jayg/AOP.
[10] John Viega, J.T.Bloch and Pravir Chandra, "Applying Aspect Oriented Programming to security", cutter IT Journal, Vol. 14, No. 2, 2001.
[11] Martin Lippert, Cristina Lopes, 1999. "A Study on Exception Detection and Handling Using Aspect-Oriented Programming", Technical Report, Xerox PARC
[12] Mercedes Amor, Lidia Fuentes, Mónica Pinto, 2006. "Coordination as an Aspect in Middleware Infrastructures". In Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software.
[13] Mik Kersten, 2005. "AOP@work: AOP tools comparison".
<http://www.ibm.com/developerworks/java/library/j-aopwork1>.
[14] Nicholas Leisiecki, 2005. "AOP@work: Enhance design patterns with AspectJ".
<http://www.ibm.com/developerworks/java/library/j-aopwork>.