Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 85 (2016) 906 – 914

Procedia Computer Science

International Conference on Computational Modeling and Security (CMS 2016)

Applying Separation of Concern for Developing Softwares Using Aspect Oriented Programming Concepts

Mr. Anil Kumar[a], Dr. Arvind Kumar[b], Mr. M. Iyyappan[a,b,]*

[a]Department of Computer Science & Engineering, Ph. D. Scholar, SRM University, Sonepat-131029, India

[b]Department of Computer Science & Engineering, Assistant Professor, SRM University, Sonepat-131029, India

Abstract

Aspects provide a means of separating cross-cutting concerns from our core implementation code into separate modules. Cross-cutting concerns are pieces of functionality that are used across multiple parts of a system. They cut across, as opposed to standing alone. For example if every method of our program requires logging information for identification then a logging aspect can be applied to methods external to the method implementation without using logging information with the methods internally. It's a powerful technique to help employ the principle of separation of concerns within code. Aspect Oriented Programming (AOP) is a methodology that provides separation of crosscutting concerns by introducing a new unit of modularization—an *aspect*. Each aspect focuses on a specific crosscutting functionality.

Keywords: Aspects, Crosscutting Concern, AOP, AspectJ.

1. Introduction

Today's software systems are complex, and all indications point to even faster growth in software complexity in the coming years. What can a software developer do to manage complexity? If complexity is the problem, modularization is the solution. By breaking the problem into more manageable pieces, we have a better shot at implementing each piece. When we're faced with complex software requirements, we're likely to break those into multiple parts such as business functionality, data access, and presentation logic. We call each of these functionalities *concerns* of the system. In a banking system, we may be concerned with customer management, account management, and loan management. We may also have an implementation of data access and the web layer.

* Dr. Arvind Kumar Tel.:+91 8295032966
E-mail address: k.arvind33@gmail.com

Peer-review under responsibility of the Organizing Committee of CMS 2016
doi:10.1016/j.procs.2016.05.281

We call these *core concerns* because they form the core functionality of the system. Other concerns, such as security, logging, resource pooling, caching, performance monitoring, concurrency control, and transaction management, cut across—or *crosscut*—many other modules. We call these functionalities crosscutting concerns.

For core concerns, object-oriented programming (OOP), the dominant methodology employed today, does a good job. We can immediately see a class such as *LoanManagementService* implementing business logic and *AccountRepository* implementing data access. But what about crosscutting concerns? Wouldn't it be nice if we could implement a module that we identify as Security, Auditing, or Performance- Monitor? We can't do that with OOP alone. Instead, OOP forces us to fuse the implementation of these functionalities in many modules. This is where aspect-oriented programming (AOP) helps.

AOP is a methodology that provides separation of crosscutting concerns by introducing a new unit of modularization—an *aspect*. Each aspect focuses on a specific crosscutting functionality. The core classes are no longer burdened with crosscutting concerns. An *aspect weaver* composes the final system by combining the core classes and crosscutting aspects through a process called *weaving*. Thus, AOP helps to create applications that are easier to design, implement, and maintain.

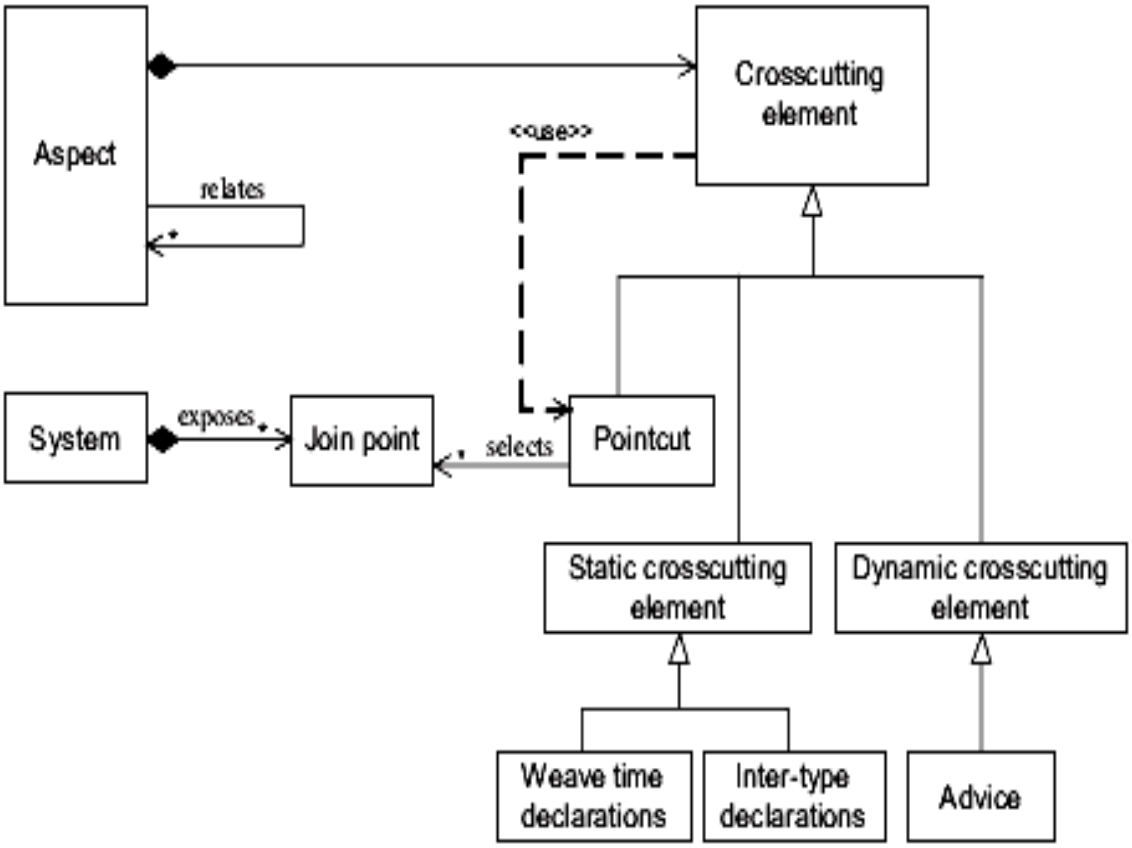

Figure 1: Generic model of an AOP system

2. Fundamental Concepts in AOP

The main benefit to using AOP is clean code that's easier to read, less prone to bugs, and easier to maintain. Making code easier to read is important because it allows new team members to get comfortable and up to speed quickly. To implement a crosscutting concern, an AOP system may include many of the following concepts:

■ *Identifiable points in the execution of the system* — The system exposes points during the execution of the system. These may include execution of methods, creation of objects, or throwing of exceptions. Such identifiable points in the system are called *join points*.

■ *A construct for selecting join points* —Implementing a crosscutting concern requires selecting a specific set of join points. The *pointcut* construct selects any join point that satisfies the criteria. This is similar to an SQL query selecting rows in database. A pointcut may use another pointcut to form a complex selection. Pointcuts also collect context at the selected points. For example, a pointcut may collect method arguments as context.

■ *A construct to alter program behavior* —After a pointcut selects join points, we must augment those join points with additional or alternative behavior. The *advice* construct in AOP provides a facility to do so. An advice adds behavior before, after, or around the selected join points. Before advice executes before the join point, whereas after advice executes after it. Around advice surrounds the join point execution and may execute it zero or more times. Advice is a form of *dynamic crosscutting* because it affects the execution of the system.

■ *Constructs to alter static structure of the system* —Sometimes, to implement crosscutting functionality effectively, we must alter the static structure of the system. For example, when implementing tracing, we may need to introduce the logger field into each traced class; *inter-type declaration* constructs make such modifications possible. In some situations, we may need to detect certain conditions, typically the existence of particular join points, before the execution of the system; *weave-time declaration* constructs allow such possibilities. Collectively, all these mechanisms are referred to as *static crosscutting*, given their effect on the static structure, as opposed to dynamic behavior changes to the execution of the system.

■ *A module to express all crosscutting constructs* —Because the end goal of AOP is to have a module that embeds crosscutting logic, we need a place to express that logic. The *aspect* construct provides such a place. An aspect contains pointcuts, advice, and static crosscutting constructs. It may be related to other aspects in a similar way to how a class relates to other classes. Aspects become a part of the system and use the system (for example, classes in it) to get their work done. Figure 1 below shows all these players and their relationships to each other in an AOP system.

3. Review of Aspect Oriented Programming

Much of the early work that led to AOP today was done in research institutions. Cristina Lopes and Gregor Kiczales of the Palo Alto Research Center (PARC), a subsidiary of Xerox Corporation, were among the early contributors to AOP. Gregor coined the term *AOP* in 1996 and started AspectJ, the first implementation of AOP. But AOP is a methodology with many possible implementations. Each implementation takes a slightly different view on the target use case and programming constructs.

3.1 AspectJ

AspectJ is the original and still the best implementation of AOP. After a few initial releases, Xerox transferred the AspectJ project to the open source community at eclipse.org. In its early implementations, AspectJ extended Java

through additional keywords to support AOP concepts, similar to the way C++ extended C to support OOP concepts. As an implementation, it provided a special compiler. Until a few years back, AspectJ had a close cousin: AspectWerkz. This AOP system followed the core AspectJ model, except that it used metadata expressed through Javadoc annotations, Java 5 annotations, or XML elements in place of additional keywords. In AspectJ version 5, AspectWerkz merged with AspectJ, offering developers a choice of technologies including a new @AspectJ (pure Java 5 annotation-based) syntax.

AspectJ's primary tool support is an Eclipse plug-in, AspectJ Development Tools (AJDT). One of AJDT's most important features is a tool for visualization of crosscutting, which is helpful for debugging a pointcut specification. Although we write classes and aspects separately, we can visualize the combined effect even before the code is deployed.

The AspectJ language has an alternative implementation called the AspectBench compiler (abc; http://aspectbench.org). The focus of this project is to provide a flexible implementation to support experimenting with new AspectJ language features and optimization ideas.

3. 2. Spring AOP

Spring is the most popular lightweight framework for enterprise applications. To satisfy the needs of enterprise applications, it includes an AOP system based on interceptors and the proxy design pattern. Earlier implementations of Spring AOP (prior to Spring 2.0) offered a somewhat complex programming model. The new programming model, based on AspectJ, offers a much better programming experience and enables Spring users to write custom aspects without difficulty. Like AspectJ, Spring AOP, through the Spring IDE (an Eclipse plug-in), provides support for visualizing crosscutting in the IDE. Spring.NET is the .NET counterpart of the Spring Framework. It includes AOP support that is similar to Spring AOP.

3.3 Other implementations of AOP

Many other implementations of AOP in Java are available. JBoss (http:// www.jboss.org/jbossaop), an open source application server, offers an AOP solution that includes a pointcut language similar to that of AspectJ. In addition, the AOP Alliance API is implemented in frameworks such as Guice (http://code.google.com/p/google-guice) and Seasar (http://www.seasar.org). (Spring used to offer a programming model based on the AOP Alliance API, but that model has been designated a transitional technology status due to the availability of the AspectJ-based model).

AspectJ has been an inspiration for AOP implementations for other languages such as Aquarium for Ruby (http://aquarium.rubyforge.org), Aspect-Oriented C (http://research.msrg.utoronto.ca/ACC), and AspectC++ (http://www.aspectc.org). Groovy, like Ruby, makes it possible to implement an AOP-like functionality through its metaobject protocol (MOP) facility (see http://www.infoq.com/articles/aop-with-groovy for an explanation of this approach). But as with Ruby, efforts are underway to introduce an AspectJ-like syntax to provide a domain-specific language (DSL) to simplify writing aspects (see http://svn.codehaus.org/grails-plugins/grails-aop for the code of the yet-to-be-released grails-aop project).

AOP has generated quite a bit of interest in the .NET world. Due to the use of byte code representations and the possibility of using proxies, .NET offers choices similar to those available in the Java world. In addition to Spring.NET, prominent AOP solutions in .NET include PostSharp (http://www.postsharp.org) and Aspect# (http://www.castleproject.org/ aspectsharp). LOOM.NET (http://www.dcl.hpi.uni-potsdam.de/research/loom) is a research project that's exploring static and dynamic weaving in .NET.

4. Implementing AOP

Aspect-oriented programming (AOP) in accomplished by implementing a series of primary concerns in a given language. These crosscutting concerns are added to the system through an aspect-oriented language. The support code developed using the aspect-oriented language is used to implement any crosscutting concerns based on common AOP terms and must be weaved into the primary application. In most implementations, the support code is written in the same language as the primary application; that is the case for AspectJ. Figure 2 shows the generalized AOP process.

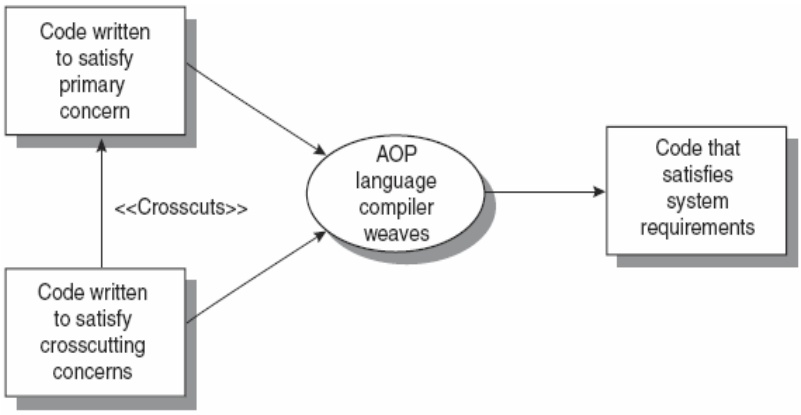

Figure 2: AOP Process

The primary goal of an AOP language is the separation of concerns. An application is written in a language that best satisfies the needs of the application and the developers. This language could be Java, C++, C#, Visual Basic, or even COBOL; in all these languages, a compiler converts the written language syntax into a format the machine can execute. In the case of Java or .NET, the language syntax is converted to byte code, which in turn is executed by a runtime environment.

4.1 AOP Language Specification

The major components of an AOP language are as follows:

- Join points
- A type of language to match join points
- Advice
- An encapsulating component, such as a class

Join Points

A *join point* is a well-defined location within the primary code where a concern will crosscut the application. Join points can be method calls, constructor invocations, exception handlers, or other points in the execution of a program. Suppose the specification document for a new system created by an AOPaware team includes a concern stating that all SQL executions to the database should be logged. To facilitate the development of the primary system, a transaction component class is created to handle all database communication from business-level components. Within the transaction component, a method called updateTables() handles all database updates. To fully implement the crosscut concern, we need to add code to the method to register a timestamp when the method is

first called. We must also include code at the end of the method to register a timestamp and add a success flag to the log. Thus, the join point to the implementation is the name of the method along with (possibly) the class name. For example, the following statement describes a join point:

public String DBTrans.updateTables(String);

The exact syntax will vary from language to language, but the goal of the join point is to match well-defined execution points.

Pointcuts

Given that the join point is a well-defined execution point in an application, we need a construct that tells the aspect-oriented language when it should match the join point. For example, we may want the aspect language to match the join point only when it is used in a call from one object to another or possibly a call from within the same object. To handle this situation, we can define a designator named call() that takes a join point as a parameter:

call(public String DBTrans.updateTables(String))

The designator tells the aspect language that the public String DBTrans.updateTables(String) join point should be matched only when it's part of a method call.

In some cases, we may use multiple designators to narrow the join point match or create groupings. Regardless, another construct called a *pointcut* is typically used to group the designators. A pointcut can be named or unnamed, just as a class can be named or anonymous. For example, in the following example the pointcut is called updateTable(). It contains a single designator for all calls to the defined join point:

```
Pointcut updateTable() :
call(public String DBTrans.updateTables(String))
```

Advice

In most AOP specifications, advice code can execute at three different places when a join point is matched: before, around, and after. In each case, a pointcut must be triggered before any of the advice code will be executed. Here's an example of using the before advice:

```
before(String s) : updateTables(s) {
System.out.println("Passed parameter – " + s);
}
```

Once a pointcut has triggered, the appropriate advice code executes. In the case of the previous example, the advice code executes before the join point is executed. The String argument is passed to the code so it can be used if needed. In most AOP systems, we have access to the object associated with the join point as well as other information specific to the join point itself.

Aspects

A system that has 10 crosscutting concerns might include 20 or so join points and a dozen or more pointcuts with associated advice. By using AOP, we can reduce code tangling and disorganization rather than create more. With this in mind, the aspect syntax was developed to handle encapsulation of join points, pointcuts, and advice. Aspects are created in much the same manner as classes & allow for complete encapsulation of code related to a particular concern. Here's an example aspect:

```
public aspect TableAspect
{
pointcut updateTable(String s) :
call(public String DBTrans.updateTables(String) &&
args(s);
before(String s) : updateTable(s) {
System.out.println("Passed parameter – " + s);
}
}
```

The TableAspect aspect is an object that implements a concern related to the UpdateTables() method. All the functionality required for this concern is neatly encapsulated in its own structure.

4.2 AOP Language Implementation

The examples presented so far are written in the AspectJ AOP language and follow the Java specification because AspectJ is designed to be used with applications written in Java. Once a concern has been written in an AOP language, a good deal of work must still be done to get the primary and AOP applications to run as a complete system. This task of integrating the crosscutting concern code and the primary application is called *weaving*.

Using Java as an example, at some point in development a number of classes and possibly aspects will represent all the concerns defined for a particular application. The primary application can be compiled into Java byte code using the Javac compiler. Once compiled, the application byte code can be executed within the Java Runtime Environment. Unfortunately, a number of aspects also need to execute. Because the aspects are Java code as well, it isn't unreasonable to think that a compiler can be used to convert the aspect code into pure Java code; the aspects are converted to classes, and pointcuts, join points, and designators are turned into other Java constructs. If this step is performed, the standard Java compiler can also be used to produce byte code from the aspects.

Assume that a compiler is available that will convert both the Java and aspect code into Java byte code during the compilation process. We need a way to incorporate the aspect code into the Java code. In compile-time weaving, the aspect code is analyzed, converted to the primary language if needed, and inserted directly into the primary application code. So, using the previous example, we know that a join point has been defined on the updateTables() method and that a pointcut defined to execute before the updateTables() method actually executes. The compile-time weaver finds the updateTables() method and weaves the advice code into the method. If the aspect is converted to a class, the call within the updateTables() method can reference a method of the new aspect object.

Here's a simple example of what the code might look like after the compile-time weaver pulls together the primary Java code and the aspect defined earlier:

```
public String updateTables(String SQL) {
//start code inserted for aspect
TableAspect.updateTable(SQL);
initializeDB();
sendSQL(SQL);
}
```

In this example, a call is inserted to the updateTable() method of the tablesAspectClass class created from the TableAspect aspect code defined earlier. This work is handled by a preprocessor before any traditional compilation takes place. Once the aspect has been weaved into the primary application code, the resulting intermediate files are sent to the Java compiler. The resulting system code implements both the primary and crosscutting concerns.

One of the downfalls of a compile-time weaving system is its inability to dynamically change the aspect used against the primary code. For example, suppose an aspect handles the way the updateTables() method connects to the database. A simple connection pool can consist of the details within the aspect. It would be interesting if the aspect could be swapped with another aspect during execution of the primary application based on predefined rules. A compile-time weaver cannot do this type of dynamic swapping, although code can be written in an aspect to mimic the swapping. In addition, compile-time weaving suggests that we need to have the source code available for all aspects, and convenience features like JAR files cannot be used.

A link-time or run-time weaver doesn't weave the aspect code into the primary application during the compile but waits until runtime to handle the weave. A processor is still used to place hooks in the methods/constructor of the primary language as well as other strategic places. When the hooks are executed, a modified runtime system determines whether any aspects need to execute. As we might expect, dynamic weaving is more complicated because of the need to change the system where the application is executing. In a byte-code system where a runtime environment is available, the process isn't as involved as a system like C++, where a compiler produces machine-level code.

5. Conclusion

Traditional languages fail in modularizing functionalities that are tangled and scattered with respect to the principal decomposition offered by the language in use. Correspondingly the source code is hard to understand, to maintain and to evolve, because the comprehensibility of the principal system decomposition is complicated by the presence of these tangled crosscutting concerns. Moreover, the comprehensibility of the crosscutting concern itself cannot be so easy, since it is scattered throughout a big portion of the system. AOP provides a better modularization of the source code, because scattered concerns can be now implemented into a single place, an aspect. In this way, many comprehensibility problems due to tangling and scattering can be addressed.

Migration towards AOP has been broken down into two steps. Aspect mining is the first step. It consists of the identification of the crosscutting concerns in the existing application. These crosscutting concerns are the aspect candidates. After identification, the actual transformation takes place in the refactoring step. The proposed techniques have been implemented in a collection of tool prototypes which are freely available. The proposed methods have been applied to a number of software systems for a total of more than half a million lines of code, in order to show the feasibility of the migration process and to evaluate the results.

References

1 AspectJ Hompage, http://www.aspectj.org/, 2001
2. S. Hanenberg, R. Unland, *Concerning AOP and Inheritance*, Dept. of Mathematics and Computer Science University of Essen.
3. G.Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, *Getting Started with AspectJ*, Communications of the ACM, Vol. 44, Issue 10, October 2001, pp. 59-65
4. C. V. Lopes, G. Kiczales, *Recent Developments in AspectJ*, Xerox Palo Alto Research Center.
5. Bergmans, L. and Aksit, M., *Composing Crosscutting Concerns using Composition Filters.* Communications of the ACM, 2001, **44**(10).
6. Rashid, A., et al. *Early Aspects: A Model for Aspect-Oriented Requirements Engineering. IEEE Joint International Requirements Engineering Conference (Accepted for Publication)*, 2002.
7. Beltagui, F.: Features and Aspects: *Exploring feature-oriented and aspect-oriented programming interactions.* Technical Report No: COMP-003-2003; Lancaster University, Lancaster, 2003
8. R. Filman and D. Friedman. *Aspect-oriented programming is quantification and obliviousness*. In Proc. Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000.
9. Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. *Role-based refactoring of crosscutting concerns.* In AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, pages 135{146, New York, NY, USA, 2005. ACM Press.
10. T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. *Discussing aspects of AOP*. Communications of the ACM, 44(10):33--38, Oct. 2001.
11. L. Blair and M. Monga 2003. *Reasoning on AspectJ programmes.* In 3rd German Informatics Society Workshop on Aspect-Oriented Software Development (AOSD-GI), (Essen, Germany).
12. AOSD, "Aspect-Oriented Software Development", http://aosd.net, 2005.

13. R. Laddad, 2003. “AspectJ in Action”. Manning Publications, Greenwich, Connecticut.