Volume 5, Issue 7, July 2015 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Migration to Aspect Oriented System

Sameer
Department of Computer Science
Shah Satnam Ji P.G. Boys' College, C.D.L.University, India

Abstract: Aspect-oriented software development (AOSD) is a new technique that improves separation of concerns in software development. AOSD makes it possible to modularize crosscutting concerns of a software system, thus making it easier to maintain and evolve. Some aspects of system implementation, such as logging, error handling, standards enforcement and feature variations are notoriously difficult to implement in a modular way. The result is that code is tangled across a system and leads to quality, productivity and maintenance problems. Aspect Oriented Software Development enables the clean modularization of these crosscutting concerns. Aspect-Oriented Programming (AOP) provides mechanisms for the separation of crosscutting concerns—functionalities scattered through the system and tangled with the base code. Existing systems are a natural test bed for the AOP approach since they often contain several crosscutting concerns which could not be modularized using traditional programming constructs. This paper presents an approach to the problem of migrating systems developed according to the Object-Oriented Programming (OOP) paradigm into Aspect-Oriented Programming (AOP). To solve this problem, AspectJ Development Tool (AJDT) 1.4.1 is used with Eclipse 3.2 as plug-in.

Key Words: Aspect-oriented software development, AspectJ, AspectJ Development Tool, Aspect Oriented Program, Cross Cutting Concern.

I. INTRODUCTION

Aspect-oriented software development (AOSD) is a new technique that improves separation of concerns in software development[2]. AOSD makes it possible to modularize crosscutting concerns of a software system, thus making it easier to maintain and evolve[4]. Research in AOSD has focused mostly on the activities of software system design, problem analysis, and language implementation. Although it is well known that testing is a labor-intensive process that can account for half the total cost of software development, research on testing of AOSD, especially automated testing, has received little attention.

Aspect-Oriented Programming is a new technology for dealing explicitly with separation of concerns in software development[5]. Just as with the introduction of object-oriented programming, aspect oriented programming brings a unique set of benefits and challenges. By far, the most compelling argument for aspect-oriented programming is that it significantly increases modularity and cohesion, thereby increasing understandability and easing the maintenance burden. In particular, AOPs use abstractions representing *concerns* that *cross-cut* the program modules that implement the primary functionality[5].For example, code that implements a particular security policy is commonly distributed across a set of classes and methods that are responsible for enforcing the policy. However, with AOPs, the code implementing the security policy can be factored out (typically into one aspect). This aspect localizes in one cohesive place the code that affects the implementation of multiple classes and methods[5].

The use of AOPs alters the development process. Classes and methods of core concerns are developed and tested as before. However, instead of embedding the code for cross-cutting actions into method bodies, separate aspects are defined that contain the code. Later, the aspects are woven into the classes that rep- resent the core concerns of the system. Once complete, the woven targets should be the composite of behavior of both core and cross-cutting concerns. The advantages of this approach include greater modularity and cohesion, a cleaner separation of concerns, and improved locality of change.

II. ASPECT-ORIENTED PROGRAMMING CONCEPTS

The core of a software-based system is the subject matter of some application domain[6]. For example, in a library, the subject matter will include *Books*, *Periodicals*, library *Patrons*, the *Catalog* of works, and so on. A system designed to manage the operation of a library must incorporate each of these subject matter items and more. Collectively, the subject matter of the system form the *core concerns* of the system and constitute the domain information that must be managed. In the implementation of the system, each core concern will typically be represented as a single abstraction with a corresponding implementation using some concrete mechanism, such as a class in an object oriented language. However, not all concerns can be represented in this discrete manner. Instead, the implementation of some concerns depend on the context provided by the behavior and concrete representation of other concerns. Such concerns are referred to as ***cross-cutting concerns***.

A basic tenet of aspect-oriented programmng is that all concerns should be treated as modular units regardless of the limitations of the implementation languages. The primary mechanism for defining solutions to cross-cutting concerns is the ***aspect***. Aspects encapsulate behavior and state of those crosscutting concerns whose implementations must span across the core concerns that form the subject matter of a system. Aspects make it possible to create cohesive modules that implement specific cross-cutting concerns that otherwise would have to be distributed across many core concerns. By placing these crosscutting concerns separately in an aspect, the core concerns are made more cohesive since their implementations are relieved of the burden of managing concepts unrelated to their purpose.

We have found many programming problems for which neither procedural not object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement. This forces the implementation of those design decisions to be scattered throughout the code, resulting in **"tangled"** code that is excessively difficult to develop and maintain. We present an analysis of why certain design decisions have been so difficult to clearly capture in actual code. We call the properties these decisions address ***aspects*** and show that the reason they have been hard to capture is that they ***cross-cut*** the system's basic functionality. We present the basis for a new programming technique, called aspect oriented programming , that makes it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code. The discussion is rooted in systems we have built using aspect-oriented programming.

2.1 Structure of the AOP-based Implementation:

The structure of the AOP-based implementation of an application is analogous to the structure of a GP (Generalized Procedure)-based implementation of an application. Whereas a GP-based implementation of an application consists of:

1. a language
2. a compiler (or interpreter) for the language, and
3. a program written in the language that implements the application.

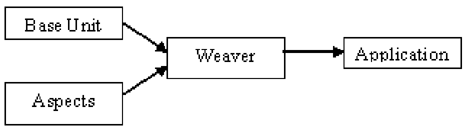

Fig.1 Weaving

The AOP-based implementation of an application consists of:

1. a *Component language* with which to program the components,
2. one or more *aspect languages* with which to program the aspects,
3. an *aspect weaver* for the combined 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).

2.2 Concepts of AOP[3]:

- **Aspect:** unit encapsulating a crosscutting concern.
- **Join point**: point in the execution of a program where an aspect might intervene.
- "*[...] whenever condition C arises, perform action A*" [3].
- – **Pointcut**: expression of a subset of join points (*condition C*)
- – **Advice**: piece of code for *action A*.
- – Pointcuts and advice encapsulated into aspects.

III. ESSENTIAL DIFFERENCES BETWEEN AOPS AND OBJECT-ORIENTED PROGRAMS:

AOPs are similar to object-oriented programs in that both have classes, interfaces, methods, packages, etc. Obviously, any Java program is a degenerate case of an AspectJ program that has no aspects[6]. Thus, the types of faults that can occur in a Java program can also occur in an AspectJ program; the source of these faults are the same as those in alternative 1 (above). New types of faults can occur as a result of using the aspect oriented features of an aspect-oriented programming language. In addition, aspect weaving can affect the types of faults commonly found in object-oriented programs (OOPs) and procedural programs. One way to identify potential sources of faults in AOPs is to examine what a programmer must do to develop an AOP.

Obviously, an AOP developer must consider additional concepts beyond those present in OOPs. The computational model of an AOP is based on a dynamic call graph resulting from method invocations. Within the execution that the call graph represents are join points where advice has been applied based on a pointcut match. To specify the patterns in a pointcut, the developer must analyze the syntactic structure of a core concern to determine those join points that must participate in the cross-cutting behavior supplied by an aspect. Based on the syntactic structure present in those join points, the aspect developer must specify an appropriate set of pointcuts and advice that, after weaving, will result in the desired behavior.

In some cases, the aspect developer must consider a wider context beyond that provided at a particular join point. This occurs, for example, when an aspect must maintain some information on a per-object basis. A common example is a cache-management aspect that must track the last access time of each object stored in the cache.

Another consideration is the flow of control within a dynamic call graph with respect to a particular join point. In certain situations, information must be collected only when the flow of control is executing within a body of code corresponding to a particular join point, such as a call to a method *m*. In this example, information is collected from the thread of execution until control returns to the caller of *m*. In other cases, information collection may be restricted to when the thread of execution has not yet returned to the caller of *m* and is not executing within the body of *m*.
In addition to execution, an aspect developer must consider what information about a join point must be passed to advice. In AspectJ, some of this information is made available automatically. For example, the predefined pointcuts *this*(), *target*(), and *args*() are used to collect information about the context of the join point. Such information includes arguments to method calls and constructor invocations, a reference to the target object, and return values.
Clearly, the unique characteristics of AOPs do not occur in object-oriented, or even procedure-oriented, programs. Each characteristic has the possibility to manifest new fault types that ultimately lead to program failures.

IV. TOOLS USED IN MIGRATION:

4.1 Eclipse 3.2[1]:

Eclipse is an open source community whose projects are focused on building an extensible development platform, runtimes and application frameworks for building, deploying and managing software across the entire software lifecycle. Many people know us, and hopefully love us, as a Java IDE but Eclipse is much more than a Java IDE.
The Eclipse open source community has over 60 open source projects. These projects can be conceptually organized into seven different "pillars" or categories:

1. Enterprise Development
2. Embedded and Device Development
3. Rich Client Platform
4. Rich Internet Applications
5. Application Frameworks
6. Application Lifecycle Management (ALM)
7. Service Oriented Architecture (SOA)

The Eclipse community is also supported by a large and vibrant ecosystem of major IT solution providers, innovative start-ups, universities and research institutions and individuals that extend, support and complement the Eclipse Platform. The exciting thing about Eclipse is many people are using Eclipse in ways that we have never imagined. The common thread is that they are building innovative, industrial strength software and want to use great tools, frameworks and runtimes to make their job easier.

What is the Eclipse Foundation?[1]

The Eclipse Foundation is a not-for-profit, member supported corporation that hosts the Eclipse projects. The Foundation provides services to run the IT infrastructure, IP due diligence, mentor the open source projects during the Eclipse development process and provide marketing and business development support for the Eclipse community.
It is important to realize that the Eclipse Foundation does not actually develop the open source code. All of the open source software at Eclipse is developed by open source developers, called committers, which are volunteered or contributed by organizations and individuals.
Eclipse projects provide tools and frameworks that span the entire software development lifecycle, including modeling, development, deployment tools, reporting, data manipulation, testing and profiling. The tools and frameworks are primarily focused on building JEE, web services and web applications. Eclipse also provides support for other languages, such as C/C++, PHP, and others.

4.2 AspectJ development Tool 1.4[1]

The AspectJ Development Tools (AJDT) project is an Eclipse Tools project which enables the development of AspectJ applications in Eclipse. The process of creating a new AspectJ project in Eclipse is remarkably similar to that of creating a Java project: Simply select File > New > Project > AspectJ Project, or use the New AspectJ Project shortcut on the toolbar. The rest of the wizard is pretty much identical to the New Java Project wizard.
In fact, an AspectJ project retains its Java nature (and has an AspectJ nature added) so that any tools and plug-ins that operate on Java projects will also operate on AspectJ projects. The key difference is the builder used to compile the project: The AspectJ compiler is used instead of the JDT Java compiler. The AspectJ compiler itself is an extension of the JDT compiler, and is therefore fully capable of compiling Java code.
AJDT is developed with AJDT. This offers a great chance to demonstrate some possible uses of aspects and also means that new versions of AJDT get a lot of manual testing (in addition to many automated tests). AJDT is implemented by a number of AspectJ-enabled plug-ins, with a total of around 200,000 lines of code. Aspects are used to implement a number of crosscutting concerns:

- catch exceptions and write appropriate entries to the Eclipse error log,
- enforce coding standards and conventions,
- take various performance and resource usage measurements in the Visualiser,
- keep track of changes across multiple properties pages

- wrap calls using Eclipse's ISafeRunnable interface (for example when calling untrusted code contributed by a plug-in extension)

4.3 Language Used:
4.3.1 AspectJ:

- Aspect-Oriented extension to Java.
- Aspect language (new constructs for aspects).
- Produces standard Java bytecode.
- Weaves into class files.

AspectJ is a language designed to support aspect oriented programming in conjunction with the Java programming language[9].AspectJ achieves modularity with an aspect abstraction mechanism which encapsulates the behavior and state of a cross-cutting concern. Unlike classes, aspects cannot be directly instantiated as objects. The activation of an aspect depends upon context provided by the core concerns represented as classes. Thus, while cross-cutting concerns are expressed as separate modular units, their definition and execution depend upon the context of a core concern's control and data flow.

4.3.2AspectJ Terminology:

- Aspect: unit encapsulating a crosscutting concern.
- Join point[3]: point in the execution of a program where an aspect might intervene.
- "*[...] whenever condition C arises, perform action A*" [3]

– Pointcut: expression of a subset of join points (*condition C*)[3]
– Advice[3]: piece of code for *action A*.
– Pointcuts and advice encapsulated into aspects

4.3.3 AspectJ Code for Comparison of two point:

```
package introduction;
public aspect ComparablePoint {
  declare parents: Point implements Comparable;
  public int Point.compareTo(Object o) {
    return (int) (this.getRho() - ((Point)o).getRho());
  }
  public static void main(String[] args){
    Point p1 = new Point();
    Point p2 = new Point();
    System.out.println("p1 =?= p2 :" + p1.compareTo(p2));
    p1.setRectangular(2,5);
    p2.setRectangular(2,5);
    System.out.println("p1 =?= p2 :" + p1.compareTo(p2));
    p2.setRectangular(3,6);
    System.out.println("p1 =?= p2 :" + p1.compareTo(p2));
    p1.setPolar(Math.PI, 4);
    p2.setPolar(Math.PI, 4);
    System.out.println("p1 =?= p2 :" + p1.compareTo(p2));
    p1.rotate(Math.PI / 4.0);
    System.out.println("p1 =?= p2 :" + p1.compareTo(p2));
    p1.offset(1,1);
    System.out.println("p1 =?= p2 :" + p1.compareTo(p2));
  }
}
```

OUTPUT:

```
p1 =?= p2 :0
p1 =?= p2 :0
p1 =?= p2 :-1
p1 =?= p2 :0
p1 =?= p2 :0
p1 =?= p2 :2
```

V. CONCLUSIONS

The complexity of software development motivated aspect-oriented programming as an approach to separate concerns and to build high quality software more easily, with higher maintainability, and a better chance at successful evolution. We believe that aspect-oriented programming has tremendous potential for building the software of the future, but only if we can pair effective and efficient testing techniques with aspect-oriented programming software construction methods. Thus we see the combination of sound AOP construction methods and systematic AOP testing as an important step in broad acceptance of aspect oriented programming as a software development approach.

We have traced the complexity in some existing code to a fundamental difference in the kinds of properties that are being implemented. Components are properties of a system, for which the implementation can be cleanly encapsulated in a generalized procedure. Aspects are properties for which the implementation cannot be cleanly encapsulated in a generalized procedure .Aspects and cross-cut components cross-cut each other in a system's implementation.

Based on this analysis, we have been able to develop aspect-oriented programming technology that supports clean abstraction and composition of both components and aspects. The key difference between AOP and other approaches is that AOP provides component and aspect languages with different abstraction and composition mechanism. A special language processor called an aspect weaver is used to coordinate the co-composition of the aspects and components.

REFRENCES

[1] AspectJ compiler 1.2, May 2004. http://eclipse.org/aspectj/.

[2] L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001.

[3] *Bruno Harbulot* ELF Developers' Forum – Manchester - October 2005

[4] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, 2001.

[5] G. Kiczales, E. Hilsdale, , J. Hugunin, M. Kersten,J. Palm, and W. Griswold. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, Budapest, Hungary, 2001.

[6] R. T. Alexander, J. Offutt, and J. Bieman. Syntactic fault patterns in OO programs. In *Proceedings of Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '02)*, Greenbelt, Maryland, December 2002.

[7] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhot´ak, O. Lhot´ak, O. de Moor, D. Sereni,

[8] R. T. Alexander, J. Offutt, and J. Bieman. Syntactic fault patterns in OO programs. In *Proceedings of Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '02)*, Greenbelt, Maryland, December 2002.

[9] T. AspectJ. The AspectJ(TM) Programming Guide, 2002.

[10] S. Clarke and J. Murphy. Developing a tool to support the application of aspect oriented programming principles to the design phase. In *Proceedings. Of the International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.

[11] G. Denaro and M. Monga. An experience on verification of aspect properties. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 186.189. ACM Press, 2002.

[12] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33.38, 2001.