Problems in Aspect Oriented Design: Facts and Thoughts

Md. Asraful Haque

Department of Computer Engineering, Aligarh Muslim University
Aligarh,U.P.-202002,India

Abstract:
The classic challenge in writing object-oriented programs (OOP) is finding the right decomposition into classes and objects. This problem arises whenever programmers need to deal with crosscutting concerns. Aspect Oriented Programming (AOP) is a well known methodology to overcome this issue by modularizing crosscutting concerns using aspects. Programmers are slowly realizing the importance of AOP since it creates cleaner code. But AOP breaks encapsulation in joint points and modifies flow control, making the source code hard to understand.AOP is not very well tested and documented and there is a lack of specific development tools. That's why it is mainly used only for maintaining the system, rather than being a good choice for developing the initial version of the system. The main goal of this paper is to increase the acceptability of AOP by offering some tips against its drawbacks.

Keywords: *Aspect-oriented Paradigm, OOP, Software Design, Crosscutting Concerns, Aspects.*

1. Introduction

AOP has been first introduced by Gregor Kickzales in 1996. Aspect-oriented programming is not the replacement of OOP rather than it is the enhancement of OOP [1][2]. It entails breaking down the system into distinct parts. Each part is called concern. All concerns are divided into two categories. The concerns which are related with the main business logic are called "core concerns". And the other concerns which capture the peripheral requirements are known as "crosscutting concerns". "Separations of concerns" (SoC) are very necessary to manage the complexity of any large system. The evolution of a software development paradigm is driven by the need to achieve a better SoC. Dijkstra suggested that the best way to achieve SoC is through modularization [3]. OOP can easily decompose core concerns into separate, independent classes by providing abstractions. But the crosscutting concerns which play a supporting role cannot be modularized into classes by OOP. AOP solves this problem. The AOP allows crosscutting concerns to be implemented separately from core concerns into aspects. An aspect is an additional unit of modularity. It encapsulates behaviors that affect multiple classes into reusable modules. AOP is a concept, so it is not bound to a specific programming language. In AOP, a project is implemented using OO language and then crosscutting concerns are dealt separately by implementing aspects. Finally, both the code and aspects are combined into a final executable form using an aspect weaver. As a result, a single aspect can contribute to the implementation of a number of methods, modules, or objects, increasing both reusability and maintainability of the code [2][4]. Fig.1 explains the weaving process. One should note that the original code doesn't need to know about any functionality the aspect has added; it needs only to be recompiled without the aspect to regain the original functionality.

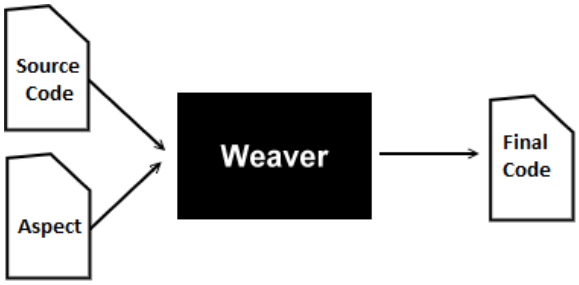

Fig. 1 Weaving process

2. Illustrative Examples

Aspect-based languages are really just aspect enhancements to current object oriented languages such as Java and C++. The main constructs added to OOP are: join points, pointcut, advive, introductions and aspects [5]. For a better understanding of these new terms, let's consider the simple example of Fig.2.


```
public class TestClass{
            public void sayHello () {
            System.out.println ("Hello, AOP");
            }
            public void sayAnyThing (String s) {
            System.out.println (s);
            }
public static void main (String[] args)
  {
   sayHello ();
   sayAnyThing ("ok");
   }
}
```

Fig. 2 TestClass.java

Now suppose, we would like to print a message before and after any call to the TestClass.sayHello() method and we need to test that the argument of the TestClass.sayAnyThing() method is at least three characters. Fig.3 is the AspectJ implementation.

```
1: public aspect MyAspect {
2: public pointcut sayMethodCall ():
   call (public void TestClass.say*() );
3: public pointcut sayMethodCallArg (String str):
   call (public void TestClass.sayAnyThing (String))
   && args(str);

4: before(): sayMethodCall() {
5: System.out.println("\n TestClass." +
   thisJoinPointStaticPart.getSignature().getName()
   + "start..." );
6: }

7: after(): sayMethodCall() {
8: System.out.println ("\n TestClass." +
   thisJoinPointStaticPart.getSignature().getName()
   + "end...");
9: }

10: before  (String str): sayMethodCallArg(str) {
11: if (str.length() < 3) {
12: System.out.println ("Error: I can't say words
    less than 3 characters");
13: return;
14: }
15: }
16: }
```

Fig. 3 MyAspect.aj

Line 1 defines an aspect in the same way we define a Java class. Like any Java class, an aspect may have member variables and methods. In addition, it may include pointcuts, advices, and introductions.

In Lines 2 and 3, we specify where in the TestClass code our modification will take place. In AspectJ terms, we define two pointcuts. To explain what a pointcut means, we first need to define join points. Join points represent well-defined points in a program's execution flow where aspects will apply [1]. Typical join points include method calls, method execution, field get and set, exception handler execution, and static and dynamic initialization. Here, we have two join points: the call to TestClass.sayHello and TestClass.sayAnyThing methods. Pointcut is a description of a set of join points based on defined criteria [1]. In our example, we define two pointcuts, named sayMethodCall and sayMethodCallArg.

An advice in AspectJ is used to define additional code to be executed before, after, or around join points. In our example, Lines 4–6 and 7–9 define two advices that will be executed before and after the first pointcut. Finally, Lines 10–15 implement an advice associated with the second pointcut and are used to set a precondition before the execution of the TestClass.sayAnyThing method. Whereas pointcuts and advice let us affect the dynamic execution of a program, introduction allows aspects to modify the static structure of a program. By using introduction, aspects can add new methods and variables to a class, declare that a class implements.

3. Problems in Aspect Oriented Design

3.1 AOP is complex.

The main challenge of software today is to manage the complexity and adaptability to the changes. Although very promising, AOP shows weakness in many dimensions, which is the consequence of lack of proper understanding and tools support. One of the main ideas of AOP is that Aspects are hidden from most developers. A problem with this is that it separates the aspects from the developers which can lead to problems. What if the developer doesn't know, or forgets, there is an aspect that will apply thread safety checking, or serialization tags? Also, what about aspect priorities when multiple aspects are applied to the same methods? Who is executed first? Are there any considerations? As a result, people use it in places where it doesn't make sense and will not use it where it does.

3.2 Debugging is hard.

The interaction of aspects with a system introduces new fault types and complicates fault resolution. Programmers rely on debugging to diagnose these faults and perform post-mortem analyses. Debugging provides a way to manually detect, diagnose, and fix anomalies and to better understand program behaviour. But debugging with aspects is tedious and painful. Aspect functionality can drastically change the behaviour and control flow of the base program, leading to unexpected result. Commercial software developers are hesitating to implement AOP-enabled products that are difficult to debug and service. The fact is debugging requires the right tools. Aspect-oriented programming is still an emerging field with many different techniques for aspect specification, composition and integration. Procedural programming feels sweet because we can understand the program flow, which maps directly to program elements. With AOP, a section of source code does not map linearly to a section of compiled code but instead maps to all instances in which a particular aspect appears. Changing a line of AOP code thus has widespread effects on compiled code. Aspect-oriented programs require extra work to map out the flow. With aop, we suddenly have code that is being run at a given point (method entry, exit, whatever) but in just looking at the code, we have no clue that it is even getting called. It happens especially when the aop configuration is in another file, like xml config. At the time of debugging an application if the advice causes some changes, things may look strange with no explanation.

3.3 Unit testing is hard.

Unit testing is a methodology for testing small parts of an application independently of whatever application uses them. Most developers today have come to agree that unit testing is good, so it's natural to want to apply unit testing to aspects [6]. But, there is no formal unit test engineering discipline established in the AOP community that provides a guide to the programmer and works to ensure some level of unit test quality. It is difficult to unit test with aspects, especially if we do the weaving at runtime. Because aspects crosscut many parts of the system, it isn't immediately clear how they can be unit tested, which has led some developers to believe they cannot be [6]. One of the hard things about testing a widespread crosscutting concern is that it can advise so many join points. Executing and checking all the matches can be a real pain. And testing for the reverse, the accidental inclusion of an unintended join point is even harder. Unit testing is a fundamental practice in Extreme Programming but most non-trivial code is difficult to test in isolation [7]. As AOP methodology is new to the programmers and it is complex, most of the times they write code which is incomplete and hard to interpret. The problem of unit testing them effectively gets harder. After all, the unit test is supposed to test the code that the programmer writes. If the programmer writes bad code to begin with, how can we expect anything of better quality in the tests?

4. Possible Solutions

4.1 Design Approach:

AOP must address both what the programmer can say and how the computer system will realize the program. The number one argument from the AOP critics is: “You cannot see what is actually going on by looking at the code”. Programmers need to be able to read code and understand what is happening in order to prevent errors. Even with proper education, understanding crosscutting concerns can be difficult without proper support for visualizing both static structure and the dynamic flow of a program [8]. So the languages which implement AOP must have the facility to support the visualizing of crosscutting concerns, as well as aspect code assist and refactoring. A good programming style and documentation also plays a vital role to make AOP simple. Few points should be kept in mind before writing the code. Joint points should be clearly exposed, aspect interface should be properly managed and the structure of the aspects should be suitable. In this context, some issues to be investigated are: (a) what are the main elements an aspect model should incorporate? (b) How can the interaction between aspect and base code be described? (c) What are the problems, conflicts, anomalies, etc. that can arise when aspect and base code are weaved? [9].

4.2 Debugging Approach:

Debugging is a critical skill that comes with common sense. AOP raises the level of abstraction [6]. Most developers have come to understand that this abstraction is ultimately for the good. Creating a layer of abstraction provides clarity and improves our understanding of what is important in a given context. And this is where a debugger comes into picture. The debugger, which understands the exact type of an object and the exact control flow, gives

linear flow to the navigation between different entities. With the right debugger, we can then examine the precise interaction between classes and aspects. Object-oriented purists also tend to overlook the clarity that aspects can bring to debugging. Debugging tools tend to fall short when certain crosscutting functionality doesn't behave as expected and it requires a tremendous amount of effort to spot all the failure points for a scattered implementation and fix them. But with recent improvements to the Eclipse AJDT plug-in, debugging aspect-oriented programs is almost as easy as debugging object-oriented ones. It uses visual tools to help us to understand and gain confidence in application's crosscutting concern. Using AJDT's cross-references view to inspect crosscutting specifications has three major advantages. First, the cross-references view gives us instant feedback as we develop our aspects. Second, it lets us easily detect consequences that would be difficult to test for. Third, the automatically generated view can verify positive cases that would be tedious to verify in code. We have to use the crosscutting comparison feature of AJDT to save a crosscutting map of our project before a refactoring or another code change. Then we will save another map after we complete the change. Finally we will compare the maps in the crosscutting comparison tool to detect any unwanted changes to the join points affected by our aspects. Note that this is an example and only AJDT provides a crosscutting comparison tool. Thinking this way, it isn't an overstatement to say that AOP actually simplifies debugging crosscutting functionality.

4.3 Unit Testing Approach:

In fact, aspects can be unit tested just as easily as classes. . In both cases, we need to break the behavior into components that we can test independently. A key concept to grasp is that crosscutting concerns divide into two different areas. First, there is the crosscutting specification, where we should ask ourselves what parts of the program the concern affects. Second, there is the functionality, where we should ask what happens at those points. With aspects, we can target one or both of these areas in isolation. Using Mock Objects for unit testing improves both base code and aspects [10]. Mock objects are objects that implement no logic of their own and are used to replace the parts of the system with which the unit test interacts. They allow unit tests to be written for everything, simplify test structure, and avoid polluting base code with testing infrastructure [7]. We can create the mock system by implementing a small subset of the classes and methods of the real application. The mock system implements just enough functionality to test the aspects. The aspect is created and tested within the mock system. To perform unit testing, we weave the aspect with the mock system, and test until we satisfy joinpoint coverage [11]. In order to properly use mock objects, a factory pattern must be used to establish a pointcut. All interactions with the service can be managed using virtual methods. Fig.4 explains the basic model.

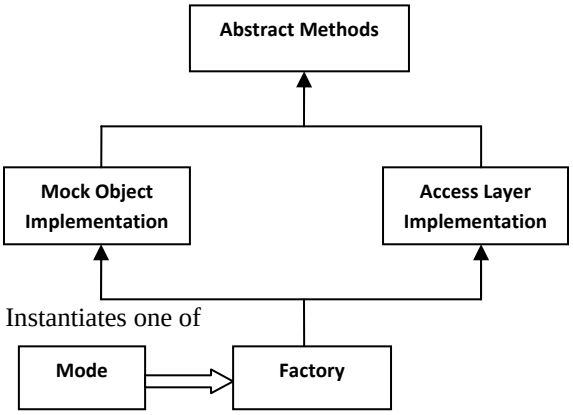

Fig. 4 Factory pattern

To achieve this construct, a certain amount of foresight and discipline is needed in the coding process. Classes need to be abstracted, objects must be constructed in factories rather than directly instantiated in code, facades and bridges need to be used to support abstraction, and data transactions need to be extracted from the presentation and business layers. These are good programming practices to begin with and result in a more flexible and modular implementation. The flexibility to simulate and test complicated transactions and failure conditions gains a further advantage to the programmer when mock objects are used. Mock systems enable aspect developers to quickly experiment with different pointcuts and advice, and iteratively develop and test aspects [11].

5. Conclusion

The current research so far in aspect-oriented software development is focused on problem analysis, software design, and implementation techniques. AOP is just too complex but still simpler than the alternatives. The advantage of using an

aspect is that code changes can be localized to the aspect even if their effects aren't. AOP complexity comes from the new mechanisms and tools used in the implementation. Developers are still confused about the technology. Some misunderstandings make it harder for developers to assess accurately whether or not to adopt AOP. Mainly the thorough knowledge and experience is the key to use this technology optimally. In this paper I have highlighted the weakness of AOP and tried to give a general idea about the solution.

Acknowledgments

First and foremost I thank to the almighty God whose blessings help me to complete this paper. I would like to thank Mr. M.R Warsi, Reader of Aligarh Muslim University and Dr. T.S. Sinha, Professor of Disha Institute, Raipur for their proper guidance and inspiration regarding the research publication. I am also grateful to Disha Institute for its support towards my higher study.

References:

[1] Pressemier N. et al., "A Safe Aspect-Oriented Programming Support for Component-Oriented Programming", ECOOP(2006)- 11th international workshop on component-oriented programming, Nantes, France.

[2] Despi I.,Luca L., "Aspect Oriented Programming Challenges", In Annals. Computer Science Series, Vol-II, fasc.-I (2008).

[3] Przybylek Adam, "Post object oriented paradigms in software development: a comparative analysis", Proceedings of the International Multiconference on Computer Science and Information Technology, pp. 1009 – 1020 (2007).

[4] Takashi Ishio, Shinji Kusumoto, Katsuro Inoue., "Debugging Support for Aspect-Oriented Program Based on Program Slicing and Call Graph", Proceedings of the 20th IEEE International Conference on Software Mai1ntenance (ICSM'2004).

[5] Qamar M.N.,Nadeem A., Aziz R., "An Approach to Test Aspect-oriennted Programs", Proceedings of the World Congress on Engineering 2007, London.

[6] Ramnivas Laddad, "AOP@Work: AOP myths and realities", http://www.ibm.com/developerworks/java/library/j_aopwork15.

[7] Mackinnon T., Freeman S., Craig P., " Endo-Testing : Unit Testing with Mock Objects", In Conference "eXtreme Programming and Flexible Processes in Software Engineering – XP2000" (Addison Wesley-2000).

[8] http://en.wikipedia.org/wiki/Aspect-oriented_software_development

[9] Chavez C.,Lucena C., "Design Level Support for Aspect-oriented Software Development", In OOPSLA'01, Work on advanced separation of concerns, 2001.

[10] Xiaofei Li Xusheng Xie, "Research of Software Testing Based on AOP", Third international Symposium on Intelligent Information Technology Application (2009).

[11] Mortensen M., Ghosh S., Bieman J., "Testing During Refactoring: Adding Aspects to Legasy Systems", Published in proc. Int. Symp. Software Reliability Engineering (ISSRE 06), pp. 221-230, 2006.

Md. Asraful Haque received his B.Tech degree in Information Technology from West Bengal University of Technology in 2007. He is presently pursuing his Master degree in Software Engineering from Aligarh Muslim University. He is a lecturer in Disha Institute of Management and Technology, Raipur, India. He has three years of teaching experience. His area of interests includes Software engineering, Operating Systems, Data Mining and Computer Networks.

