Aspect-Oriented Modeling: Issues and Misconceptions

Saqib Iqbal
Department of Informatics,
University of Huddersfield,
Huddersfield, HD1 3DH, UK
+44 (0)7588 560414
s.iqbal@hud.ac.uk

Gary Allen
Department of Informatics,
University of Huddersfield,
Huddersfield,HD1 3DH,UK
+44 (0)1484 472152
g.allen@hud.ac.uk

Abstract—**Aspect-oriented programming is an extension to object-oriented programming. It introduces new constructs called "aspects" for representing crosscutting concerns in a system development. These constructs are somewhat similar to object-oriented "classes" but they also have some clear differences in terms of characteristics. This complicated nature of aspects makes their modeling a difficult task. While working on the modeling of new design techniques for aspect-oriented technology, we have come across some unresolved modeling issues and some misconceptions about the nature of aspects and their representation in software design. This paper highlights these misconceptions and outlines some important aspect-oriented modeling issues, such as the modular nature of aspects, their resemblance with classes, and their high coupling with the base program.**

Keywords-component; Aspect-Oriented Design, Aspect-Oriented Modeling, Aspects, Software modularity.

I. INTRODUCTION

Aspect-oriented programming (AOP) [1] can be considered as an extension to object-oriented programming. Object-oriented programming fails to address the handling of so called "cross cutting concerns" of the system adequately, which leads to certain code being "scattered" over multiple classes. Apart from the implicit redundancy of such code, scattered code can also lead to inconsistencies when updating the implementation of these concerns or the wider system. Such concerns could be non-functional requirements of the system, like persistence, performance, fault-tolerance, etc., or they could be application-dependent functional requirements. AOP [1] was introduced to handle such concerns separately from their interacting base classes. AOP implements crosscutting concerns in separate class-like constructs, and they interact with the base program on well-defined control points (called join points) during the execution of the program. Such separate implementation of these scattered concerns is intended to make the system more consistent, traceable and understandable.

Since the advent of AOP, a lot of design and modeling approaches have been suggested [5] [6] [7] [8] [9] [11]. We can broadly categorize these approaches in two styles of modeling, First style suggests design notations and design modeling techniques for aspects and their elements, and the second style suggests capturing crosscutting concerns in new modular constructs such as themes [5], subjects [8] and features [9]. Most of the approaches in the former style have extended the UML notations to represent aspects and their elements, and they have adopted AspectJ [1] as their base technology. AspectJ is a technology for the development of systems in the aspect-oriented (AO) paradigm. Since AspectJ implements aspects in class-like body structures, so these techniques consider aspects analogous to classes. Similarly such approaches consider aspect elements such as advices and pointcuts analogous to classes' operations. This paper discusses the misconceptions involved in considering aspects like classes and elements like operations. We have also assessed properties of aspects to find out if they are really modular constructs, which we believe is another misconception about aspects. Finally we have suggested that aspects must not be handled as object-oriented constructs, but rather separate and well-defined design paradigms and meta-models must be developed to represent and design aspects along with the base program.
Section II discusses aspect-oriented modeling issues and some related misconceptions and section III summarizes the purpose of paper and calls for open debates on the highlighted problems.

II. AO MODELING ISSUES AND MISCONCEPTIONS

While working on developing new modeling and design strategies for aspects and their elements, we have come across some unresolved modeling issues. We have also observed difference of opinion among researchers on some fundamental definitions of aspect's abstraction and aspect's nature.

Following sub-sections discuss some of such issues and misconceptions about aspects, their behavior and structure of their elements.

A. *Aspect as Classes*

Modeling strategies [3] [4] [6] proposed so far for AOP, especially based on AspectJ, view and treat aspects as classes. It is intuitively obvious to look at aspects as classes because of their representation in AspectJ. They are declared like classes and contain the same kinds of constructs, such as attributes and operations. Figure 1 shows the similarity between classes and aspects. If we look at both constructs, we can see that an aspect's body is similar to that of a class. Aspects are declared like classes and they can have operations and attributes just like classes.

```
aspect aspect-name {                    | class class_name {
   attribute1;                          |    attribute1;
    attribute2; ...                     |    attribute2; ...
    operation1(..){..}                  |    operation1 (..) {..}
  pointcut pointcut_name (..) :         |    operation2(..) {..}
 target (..) && call (..)               |  }
advice after() : pointcut_name          |
{...}                                   |
 }                                      |
```

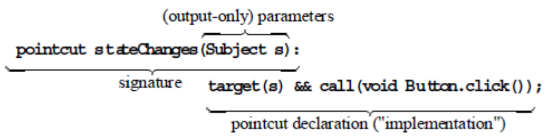
Figure 1. Code styles of aspects and classes.

Since AspectJ is built on Java technology, so it seems that the creators of AspectJ have tried to keep the representation of aspects and classes as similar as possible. But the question is; can we consider aspects as classes? The answer is not simple. Classes are pure object-oriented elements. They are fundamentally encapsulating, inheritable and instantiable constructs. If we assess aspects based on these properties, we first of all find aspects contradicting the basic principle of encapsulation (or data-hiding). Aspects do have their own data but they also access other classes' private data to perform their functionality. For example, Security and Logging aspects need to access the private data of the interacting base classes, which is a clear violation of object-oriented encapsulation. Secondly, Inheritance can partially be implemented in aspects. Aspects can have child aspects but child aspects cannot override advices of the parent aspect because parent aspect's advices do not have unique signatures or identifiers. Finally, instantiation of aspects is not similar to that of classes either. Aspects are instantiated on need, not on demand like classes and objects. Their instantiation cannot be coded within the program; rather their instantiation depends on defined control points (join points) during the execution of the program. This dynamic nature of aspects' instantiation again contradicts the behavior of classes and objects.

Now we will look at the similarity of pointcuts and advices to operations, as suggested by some of the modeling approaches like [3] [4] [6].

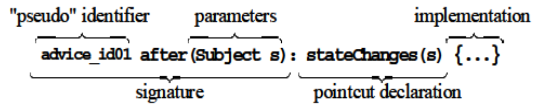
B. *Considering Pointcuts and Advices as Operations*

As suggested by [6], aspect's elements, such as pointcuts and advices, have a similar structure to a class's operation. First, we look at the pointcuts' structure. They argue (shown in Figure 2 and Figure 3) that pointcuts have a signature, they can have an arbitrary number of parameters, and they have an implementation just like an operation's structure in a class.



```
pointcut stateChanges(Subject s):
                       target(s) && call(void Button.click());
```

Figure 2. Similarity of pointcut to operations by [6].



```
advice_id01 after(Subject s): stateChanges(s) {...}
```

Figure 3. Similarity of advice to operations by [6].

In [2] and [6], which is shown in Figure 2 and Figure 3, It has been suggested that we can resemble pointcuts and advices with operations by looking at their declarations, but we argue that we have to consider properties of advices and pointcuts as well while asserting similarities with operations. Pointcuts cannot return values like operations. They have parameters passed by the base classes to establish a join point, but there is no need of returning any type which is contrary to operations (a problem also pointed out in [6]). Secondly, pointcuts cannot have local data variables; the reason behind this is that they do not process anything. They are merely used to represent join points as predicates in the program. Now looking at advices, they also have some remarkable behavioral differences to the class's operations. First, they do not have unique and identifiable signatures. This is the reason that aspects do not allow overridden advices in the child aspects. Second, they are dependent on the declaration of a corresponding pointcut.

C. *Cohesiveness and Dependency of Pointcuts on Base Program*

In this section we will talk about another AO modeling issue that is the dependency of pointcuts on the base program elements. Pointcuts are tightly coupled with the relevant base program through join points. Join points are hooks or insertion points where a piece of code is supposed to run. These points are identified by their predicates defined in pointcuts. Figure 4 can give a clear idea about the degree of coupling between pointcuts and the related base code. Aspect Change has two predicates (join points) which indicate the points in the base program where advices will be inserted. These predicates are defined with the class names and method names which makes them tightly coupled with the base code.

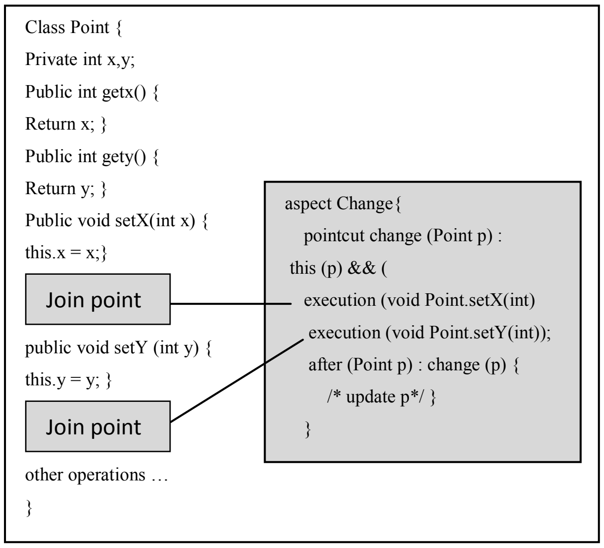

```
Class Point {
Private int x,y;
Public int getx() {
Return x; }
Public int gety() {
Return y; }
Public void setX(int x) {
this.x = x;}
Join point
public void setY (int y) {
this.y = y; }
Join point
other operations …
}
```
```
aspect Change{
   pointcut change (Point p) :
 this (p) && (
   execution (void Point.setX(int)
    execution (void Point.setY(int));
    after (Point p) : change (p) {
       /* update p*/ }
    }
```

Figure 4. Coupling of aspects with related base code.

This problem was initially pointed out by St¨orzer in [10], and he called it a "fragile pointcut problem". Any change to the base program or a refactoring process can force the designer to change pointcuts of the related aspects. Pointcuts' cohesive nature is a big hurdle in adopting or reusing them in other programs. Some efforts have been made to address this problem by introducing aspect patterns [11]. Aspect patterns can make aspects reusable but developing patterns for application-specific aspects is a big challenge. Aspects can be as general as performance, persistence or fault-tolerance, or they can be as application-dependent as a special feature of an application. Categorizing these aspects may result in so many aspect patterns that their adoption on every application may make system design a lengthy and complex process.

D. Modularity

The most controversial statement made in favor of AOP is "it improves modularity of the system". We think that this statement is made in mainly two contexts, one "it improves modularity of the system" and second "it modularizes crosscutting concerns". We talk about the first context first. If we look at the implementation style of AspectJ and composition filters, they both implement crosscutting concerns (aspects) along with object-oriented base programs. This means that, except for the crosscutting concerns' implementation, the whole system is implemented in an object-oriented way. For making a system modular, we need to modularize the whole system and all its business logic. In this case we only "modularize" the crosscutting aspects of the system, and leave the rest of the system as it is. Such a partial modularization of a system cannot provide the required results of modularity.

Now if we look at the second context, which is "AOP modularizes crosscutting concerns". This is also a confusing statement. We discuss this by analyzing the modularization of crosscutting concerns in the light of the modularization principles as discussed in [12]:

- **Modules should be opaque to the rest of the system and initialized through a well-known interface.**

  Crosscutting concerns' implementation in AspectJ and composition filters cannot be said to be opaque to the rest of the system. It fundamentally depends on the related modules of the base program (classes). Aspects are tightly coupled with the base program through join points, and they have an unbreakable connection with each other. Nor is there any such interface implementation which could make aspects reusable.

- **Modules should not directly reference one another or the application that loaded them.**

  There is a direct reference between an aspect and its corresponding base program through join points. An aspect's advices entirely depend on these references to run. So the implementation of aspects goes against this property as well.

- **Modules should use services to communicate with the application or with other modules.**

  A modular implementation of a system will stress the need for an intermediary application layer which should be responsible for any communication among modules and between modules and the application. In AOP we don't see any such layer because communication is done directly between aspects and their corresponding classes.

- **Modules should not be responsible for managing their dependencies. These dependencies should be provided externally, for example, through dependency injection.**

  This property is designed to ensure the reusability of the modules. If the dependencies are managed within modules, it increases their coupling with their corresponding dominant or dependent modules. An aspect's implementation in AOP contradicts this property of modularization as well as aspect's despondencies are managed locally in form of pointcuts.

- **Modules should support being added and removed from the system in a pluggable fashion.**

  As a result of the issues discussed above it is very unlikely that an aspect can be unplugged from a system in a reusable fashion.

The findings above call for a renewed discussion on the assertions made in favor of modularization of a system by AOP. We believe that we should use the word "modularization" carefully when discussing separation of concerns. Concerns are separated from the base program and are handled separately before being weaved again with the base program, but they are not modularized because they are part of the base program. Their implementation is required to be along with the relevant base code. It is just that we implement them differently to make the program code easier to understand, design, implement and document.

III. Conclusion and Discussion

AOP is a new field and there are a number of processes, from requirement gathering to implementation of aspects, which need maturity. Absence of a uniform development approach for aspects has resulted in aspects being handled differently in all phases of development. Such approaches make aspects inconsistent and less traceable. AOSD requires a mature modeling technology like UML which could uniformly capture, design and implement aspects. This paper points out some of the modeling issues which one has to consider while developing design techniques for aspects. These modeling issues include treating aspect like classes, the cohesive nature of pointcuts, and the less reusable nature of aspects due to their lack of separate design and implementation technologies. Some core misconceptions about aspects have also been mentioned which have evolved over the time due to the absence of mature AOSD approaches.

The most likely solution to overcome these modeling problems and to eradicate the misconceptions about AOP is a uniform AOSD design and implementation approach. Such an approach must provide a consistent way of capturing and designing aspects. Aspects must be able to be mapped from the design phase to the implementation phase, which will only be possible if we have a specialized implementation technology for aspects.

References

[1] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming" in XEROX PARC Technical Report, 1997, SPL97-008 P9710042.

[2] Stein, D., Hanenberg, and S., Unland, R., "Designing Aspect-Oriented Crosscutting in UML" Proc. 1st International Workshop on Aspect-Oriented Modeling with UML, AOSD 2002 at Enschede, the Netherlands.

[3] Hridesh Rajan and Kevin J. Sullivan, Classpects, "Unifying aspect- and object-oriented language design" Proc. The 27th international conference on Software engineering, (St. Louis, MO, USA, 2005).

[4] T. Aldawud, and A. Bader, "UML profile for aspect-oriented software development" Proc. The Third International Workshop on Aspect Oriented Modeling. (Boston, USA, March 17-21, 2003).

[5] E. Baniassad and S. Clarke, "Theme: An Approach for Aspect-Oriented Analysis and Design" Proc. The 26th International Conference on Software Engineering, 2004.

[6] D. Stein, S. Hanenberg, and R. Unland, "A UML-based Aspect-Oriented Design Notation For Aspect" Proc. Aspect-Oriented Software Development (AOSD 2002). (Enschede, The Netherlands, 2002).

[7] D. Wagelaar and L. Bergmans, " Using a concept-based approach to aspect-oriented software design" Proc. Aspect-Oriented Design workshop (held with AOSD 2002), (Twente, Enschede, The Netherlands, 2002)

[8] S. Clarke, W. Harrison, H. Ossher, and P. Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code" Proc. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 1999) (Denver, Colorado, USA, 1999).

[9] Nebulon, 2005. http://www.featuredrivendevelopment.com/, accessed on December 14, 2009.

[10] M. St¨orzer, "Analytical problems and AspectJ" Proc. The 3rd German Workshop on Aspect-Oriented Software Development. (Essen, Germany, March 2003).

[11] James Noble, Arno Schmidmier, David J. Pearce and Andrew P. Black, "Patterns of Aspect-Oriented Design" Proc. European Conference on Pattern Languages of Programs. (Irsee, Germany, 2007).

[12] Modularity. Composite Application Guidance for WPF and Silverlight - October 2009, retrieved from http://msdn.microsoft.com/en-us/library/dd490825.aspx, accessed on January 12, 2010.