Pointcut Design with AODL

Saqib iqbal
Department of Informatics
University of Huddersfield,
Huddersfield, HD1 3DH, United Kingdom
s.iqbal@hud.ac.uk

Gary Allen
Department of Informatics
University of Huddersfield,
Huddersfield, HD1 3DH, United Kingdom
g.allen@hud.ac.uk

***Abstract*—The designing of pointcuts is a crucial step in Aspect-Oriented software development. Pointcuts decide the places where aspects interact with the base system. Without designing these pointcuts properly, the weaving process of aspects with the base system cannot be modelled efficiently. A good design of pointcuts can ensure proper identification of join points, clear representation of advice-pointcut relationships and overall efficiency of the weaving process of the system. The existing approaches do not design pointcuts separately from their parent aspects, which hinders in identifying pointcut conflicts before the implementation of the system. This paper provides a set of graphical notations to represent join points, pointcuts, advices and aspects. A graphical diagram has been proposed that shows the relationships between pointcuts and their relevant advices. The paper also provides a technique to represent and document pointcuts along with their related advices and corresponding base elements in a tabular way. The technique can help in resolving two of the most complicated problems of pointcut modelling, the fragile pointcut problem and the shared join point problem.**

Keywords-component; Aspect-Oriented Design, Pointcut Design, Pointcut Modelling, Aspect-Oriented Design Language

I. INTRODUCTION

The handling of concerns is extremely important for critical, distributed and complex systems. Each concern has to be identified, specified and designed properly to avoid inconsistencies which can lead to serious consequences if the system is critically sensitive. Object-oriented design approaches have been the pick of the design techniques for such systems during the last three decades. Unified Modelling Language [2] emerged in the 1990s, and rapidly became accepted as the standard analysis and design approach for object-oriented development of systems. Unfortunately, object-oriented approaches started showing problems in capturing concerns that are scattered in nature and whose implementation overlaps other concerns and/or system units. Such concerns are known as crosscutting concerns. Examples of crosscutting concerns include system security, logging, tracing and persistence. The implementation of these concerns resides within the implementation of other concerns or classes, which results in inconsistencies and modification anomalies.

Aspect-Oriented Programming (AOP) [1] was introduced to rectify this problem. AOP introduced a new construct, called an aspect, besides the traditional object-oriented classes. The aspect is identified, specified and designed separately, and is woven into the base system at run-time wherever it is required. AOP was proposed as an implementation solution to the crosscutting problem. That is the reason why initial developments in AOP were mainly in the field of implementation. AspectJ, AspectWerkz and JBoss AOP are among a number of implementation technologies which were proposed immediately after AOP came into existence. With the passage of time research was extended to earlier phases of development as well, resulting into development of aspect-oriented requirement engineering and design approaches. Although modularity of aspect-oriented systems has been ensured with the introduction of such techniques, the cohesive nature of aspects with the base system has not been addressed properly in the existing approaches. Pointcuts, which are responsible for identifying join points in the system where aspects are invoked, are highly dependent on the base program's structural and behavioral properties. This characteristic makes pointcuts fragile with respect to any changes in the base program, and results in a problem called *Fragile Pointcut Problem* [6,7]. Another problem related to pointcuts is *Shared Join Point Problem* [4], where more than one advice from a single or multiple aspects are supposed to be executed at a single join point. A proper design is required to resolve the precedence of advices so that they run in a pre-determined order. This paper addresses these two pointcut problems and proposes a diagrammatic and tabular approach to help in rectifying both issues without compromising the consistency of the system. The tabular approach promises better documentation of pointcuts and enables modifications to be made in a consistent manner.

The rest of this paper is structured as follows: Sections 2 and 3 describe the Fragile Pointcut Problem and the Shared Join Point Problem respectively. Section 4 describes the proposed pointcut-advice diagram and Pointcut Table, and section 5 provides discussion and conclusion of the paper.

II. THE FRAGILE POINTCUT PROBLEM

Pointcuts are considered fragile because their definitions are cohesively coupled with the constructs in the base system. Upon modification in the base system, pointcuts' semantics are bound to change [6,7]. Pointcuts are defined by join points which are specific points in the base system. Once a join point is modified in the base system, the relevant pointcut is altered to adopt that change or lose that particular join point. This fragile nature of pointcuts forces designers to reflect changes in the pointcut definitions when they make any modification to the base system. Join points are formed not only on structural characteristics of the system but also on behavioural properties of functional units of the system. Therefore, any type of change to structural or behaviour property of the base system would

require related pointcuts to be altered [3]. The fragility of pointcuts leads to two core problems: unintended join point capture problem, which arises when a join point is accessed which does not exist anymore because of modifications to the source code; and accidental join point miss problem, which happens when a join point is not captured which was originally supposed to be captured, again due to an alteration to the source code of the base program [3].

III. THE SHARED JOIN POINT PROBLEM

Aspects superimpose their behaviours at well-defined join points in the base system. A shared join point is a point where multiple aspects interact and superimpose their advices. The problem arises in deciding which aspect should run first. This is a critical decision because execution of one aspect's advice can change the attributes which are supposed to be used by another aspect's advice. Figure 1, taken from [4], illustrates the problem.

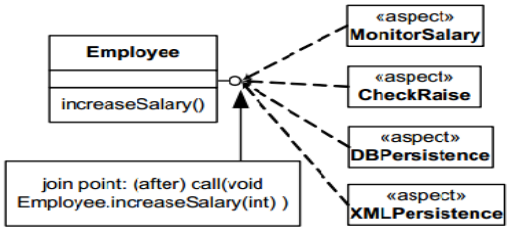

Figure 1. 'Employee' class and its superimposed aspects (taken from [4]).

This problem is considered to be an implementation-level problem and has been addressed by renowned aspect-oriented programming techniques. For example, AspectJ [10] provides a *declare precedence* keyword to order advices, Composition Filters [8] provides *Seq* operator to declare precedence, and JAC [9] determines the order by implementing wrappers in the classes which are filed in a wrapper file in an execution sequence.

IV. DESIGNING POINTCUTS IN AODL

Pointcuts rely heavily on the lexical structure of the base program. The definition of pointcuts (group of join points) contains direct reference to the syntax of base elements. This tightly coupled nature makes it hard for programmers to make any changes to base program without having knowledge of pointcuts and vice versa. Aspect-Oriented Design Language (AODL) [5] presents a diagrammatic approach to the design of pointcuts and a tabular way of documenting their definitions. This kind of well-documented representation of aspects in the design phase makes it easier for designers as well as programmers to evolve either aspects or the base program. Before moving onto the proposed models, we introduce AODL briefly in the following section.

A. *Aspect-oriented Design Language*

Aspect-Oriented Design Language (AODL) [5] is a UML-like design language that has been developed by the authors to design aspects, aspectual elements, and object-aspect relationships. AODL offers a unified framework to design both aspects and objects in one environment. The constructs of aspects are denoted by specialized design notations and structural and behavioral representations are done with the help of design diagrams. The details about the semantics of the notations can be found in [5]. Figure 2 shows design notations adopted by AODL.

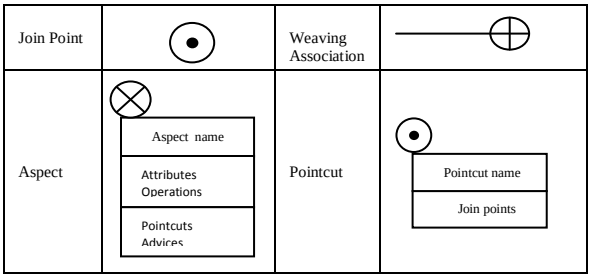

Figure 2. AODL Design Notations

AODL proposes a three phase design for aspects, constituent elements and the relationships between aspects and objects, details can be found in [5]. In the first phase, join points are modeled with the help of two diagrams, one for the structural identification of join points, known as a Join point Identification diagram, and the second for the behavioral representation of join points, known as a Join Point Behavioural diagram. In the second phase, aspects are designed with the help of an Aspect Design Diagram. And in the final phase, aspects are composed with the base model with the help of two diagrams, an Aspect-Class Structure Diagram and an Aspect-Class Dynamic Diagram. Complete details about the semantics of the language and its usage can be found in [5].

B. *Pointcut-Advice Diagram*

Pointcuts are highly dependent on the base objects through join points. Similarly, advices are tightly coupled with their corresponding pointcuts. To represent these cohesive relationships, we need to show related pointcuts, advices and base objects in a diagrammatic model. An Aspect Design Diagram (shown in Figure 3) contains the properties and behavior of an aspect. The diagram connects with the base classes through use of the crosscuts stereotype. Pointcuts are represented along with their corresponding advices and occurrence attributes (before, after and around) in a pointcut-advice diagram.

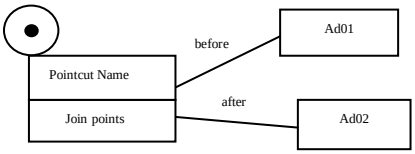

Figure 3. Pointcut-Advice Diagram

C. *Pointcut Table*

Pointcuts are predicates which are set on defined points (join points) in the execution of a program. To define and document pointcuts properly ensures consistency of the program. AODL has proposed a pointcut table (shown in Table 1) to document pointcuts along with their related advices, aspects and base classes. The table defines pointcuts in vertical columns by indicating the join points of the base system horizontally. The columns of the table provide list of aspects and complete definition of their pointcuts along with their related advices. The rows, on the other hand, show the base

system attributes, methods and execution points where join points have been identified. The execution order of advices on a single join point is declared in the last column, named Order. The table has been tested and verified to represent all types of legitimate AspectJ pointcuts, as defined in [11].

Table 1: Pointcut Table

	Aspect A		**Aspect B**		**Order**
	AdA1 (Before)	**AdA2 (After)**	**AdB1 (Before)**	**AdB2 (Around)**	
Class A	this				
Attribute					
constructor	execution				
Method1	execution		execution		AdA1, AdB1
Method2					
getX()					
Class B					
Method1	call		within		AdB1, AdA1
Method2		exception(type)			
Pointcut Definition	**this**(A) && **exec**(MA1) && **call** MB1)	**exception**(type)	**call**(A1) OR **within**(MB1)		
Pointcut	**P1**	**P2**	**P3**	**P4**	
Pointcut Trigger	**!(P2)**		**cflow(P1)**	P1 && P2	
Complete Definition	**this**(A) && **exec**(MA1) && **call** (MB1) &&!(P2)	**exception**(type)	**call**(A1) OR **within**(MB1) && cflow(P1)	P1 && P2	

In case of system being too complex and containing a number of aspects, the table can be broken into multiple tables and a specific number of aspects can be contained in each table. This way each table will contain pointcut information about a specific number of aspects only (say 3 or 5) and the readability of the system will improve.

D. Implications of the Approach

The existing aspect-oriented design approaches do not provide means to design pointcuts separately from their parent aspects. The composition of aspects heavily depends on consistency of pointcuts because they define the join points where aspects are woven into the system. The authors felt that (i) the pointcuts should be designed properly with the help of designated design notations and design diagrams, (ii) the pointcuts should be documented properly so that their features and relationships are specified efficiently before being implemented, and (iii) the pointcuts should be ordered at the modelling level so that problems such as the fragile pointcut problem and the shared join point problems are handled before implementation.

The authors of the paper do not claim that the proposed diagrammatic and tabular approach resolve both the problems completely. It is, however, suggested that adopting this type of approach can help in resolving a number of problems especially inconsistencies and conflicts involving pointcuts.

E. Example

To make the proposed approach more understandable, we will implement the *Observer Pattern* [12] as implemented by [13]. An abstract aspect Observer is extended by two aspects, *Observing1* and *Observing2* as shown in Figure 4. The following sections implement this example using a Pointcut-Advice Diagram and a Pointcut Table.

```
OBSERVER PATTERN
abstract aspect Observer {
void notify() { ... }
abstract pointcut p();
abstract pointcut c();
after(): p() {notify();}
after(): c() {notify();}
}
aspect Observing1 extends Observer {
pointcut p(): call(void Buffer.put(int));
pointcut c(): call(void Buffer.get());
}
aspect Observing2 extends Observer {
pointcut p():within(Buffer) && call(* put*(*);
pointcut c():within(Buffer) && call(* get*(*));
}
```

Figure 4: Observer Pattern Implementation (Taken from [13])

Pointcuts are designed with the help of pointcut-advice diagrams, where each pointcut is represented along with its corresponding advice. As shown in Figure 5a, pointcuts p() and c() of the *Observing1* aspect have been represented with the help of a pointcut diagram which shows the definition of the pointcut (set of join points) and the advices with the occurrence attribute (which is after for both the pointcuts). Similar diagram has been drawn in the Figure 5b for *Observing2* aspect.

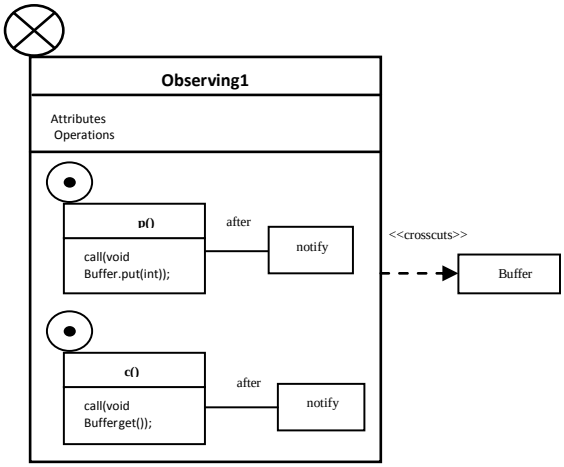

Figure 5a: Aspect Design Diagram for *Observing1*

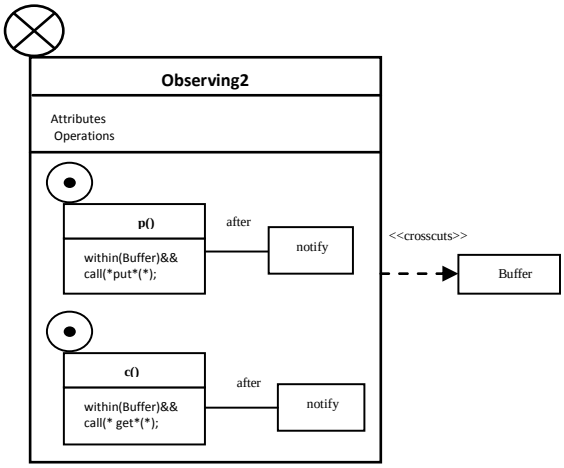

Figure 5b: Aspect Design Diagram for *Observing2*

Table 2 shows the pointcut table for the Observer Pattern. It documents all of the information about the pointcuts of a particular aspect, along with the corresponding base methods and attributes, in a tabular way. Besides documenting all the information about a pointcut, the table also shows the order in which advices are executed on a particular join point. For instance, in the first row of the table we have two advices, AdOb2_1 and AdOb2_2, which are supposed to be executed on a join point "within" which is defined on objects of the Buffer class. The table also provides the Order column, where we can show which advice should be executed first, which in the case of Table 2 shows that AdOb2_1 will execute before AdOb2_2.

Table 2: Pointcut Table for Observer Pattern

	Observing1		**Observing2**		**Order**
	AdOb1_1 (After)	**AdOb1_2 (After)**	**AdOb2_1 (After)**	**AdOb2_2 (After)**	
Class Buffer			within	within	AdOb2_1, AdOb2_2
Attributes					
put()			call		
put(int)	call		call		AdOb1_1, AdOb2_1
get()		call		call	AdOb1_2, AdOb2_2
get(int)				call	
Pointcut Definition	call(void Buffer.put(int))	call(void Buffer.get())	witin(Buffer) && call(*put*(*))	witin(Buffer) && call(*get*(*))	
Pointcut Name	P()	c()	p()	c()	
Pointcut Trigger					
Complete Definition	call(void Buffer.put(int))	call(void Buffer.get())	witin(Buffer) && call(*put*(*))	witin(Buffer) && call(*get*(*))	

V. Fragile Pointcut and Shared Join Point Problem

The method to document pointcuts, shown in Table 1 and Table 2, reduces the fragile nature of pointcuts. The table provides complete information about a pointcut, which helps in modifying the pointcut without allowing inconsistencies. This kind of tabular documentation helps remove modification anomalies when join point definitions are altered in the base system.

The pointcut table also provides a column named Order to declare the precedence of advices which have execution clashes with each other. Advices are grouped in order of their execution on a join point which is shown in that particular row. The table therefore provides an opportunity for designers to set the precedence on the execution of advices during the design phase in order to avoid clashes in execution.

It is again stressed that the authors do not claim that the proposed approach resolves all types of pointcut conflicts including the fragile pointcut problem and shared join point problem. It is, however, asserted that this approach can help in designing pointcuts at modelling level and it can help in identifying and resolving some key issues of pointcuts before they are implemented.

VI. Discussion and Conclusion

Aspect-Oriented Development unifies separately defined aspects with objects of the base system through well-defined join points. The composition between aspects and objects depends heavily on the identification of join points and their proper grouping in the form of pointcuts. The cohesive nature of pointcuts is defined by definition of join points which are susceptible to change if an alteration is made in the base program. The resultant pointcuts are inconsistent with the system and result in either missing some join points or capturing join points which are not intended by the designer. To avoid such problems, proper documentation and proper design of a pointcut becomes vitally important. This paper has proposed a diagrammatic approach to the design of pointcuts along with their related aspects, advices and base objects. The diagram does not only help structural design of a pointcut but also helps in representing the relationships between an aspect and its corresponding advices. The paper also proposes a tabular approach to the documentation of pointcuts during the design phase, so that all of the corresponding definitions of a pointcut are defined along with its characteristics and parent entities (aspects and classes). The pointcut table promises the rectification of two of the most common problems of pointcut design, the fragile pointcut problem and the shared join point problem during the design phase.

The future research will strive to develop pointcut composition models to identify aspect interferences at the pointcut level. Tool support will also be provided to automate development of pointcut models and generation of code.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. "Aspect-Oriented Programming," In: Proceedings of ECOOP 1997, Jyväskylä, Finland, June 9-13, 1997, pp. 220-242.

[2] "Unified Modelling Language", OMG, http://www.uml.org/. Accessed on 09 Dec. 2011.

[3] A. Kellens, K. Gybels, J. Brichau, and K. Mens. A model-driven pointcut language for more robust pointcuts. In Proc. Workshop on Software Engineering Properties of Languages for Aspect Technology (SPLAT), 2006.

[4] I. Nagy, L. Bergmans, and M. Aksit. "Composing aspects at shared join points". In Proceedings of International Conference NetObjectDays (NODe), Erfurt, Germany, Sep 2005.

[5] S. Iqbal and G. Allen, *Designing Aspects with AODL.* International Journal of Software Engineering. 2011, ISSN 1687-6954 (In Press)

[6] C. Koppen and M. Stoerzer. Pcdiff: Attacking the fragile pointcut problem. In First European Interactive Workshop on Aspects in Software (EIWAS), 2004.

[7] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In 21st IEEE International Conference on Software Maintenance (ICSM), pages 653–656, 2005.

[8] "Compose* portal", http://composestar.sf.net, Accessed on 24 Jan, 2012.

[9] "Java Aspect Component", http://jac.ow2.org/, Accessed on 24 Jan, 2012.

[10] Eclipse, AspectJ, http://www.eclipse.org/aspectj/, Accessed 05 Dec. 2010

[11] Pointcuts, Appendix B. Language Semantics, http://www.eclipse.org/aspectj/doc/next/progguide/semantics-pointcuts.html, Accessed on 30 Jan, 2012.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley, Reading, Ma, USA, 1995.

[13] W. Cazzola, S. Pini, and M. Ancona. Design-Based Pointcuts Robustness Against Software Evolution. In Proceedings of RAM-SE'06 Workshop, Nantes, France, July 2005