Managing Conflicts in Aspect-Oriented Software

Jane L. Pryor

ISISTAN Research Institute
Facultad de Ciencias Exactas, UNICEN
Paraje Arroyo Seco, B7001BBO Tandil, Argentina Tel/Fax: + 54 – 2293 – 440362 / 3
http://www.exa.unicen.edu.ar/~isistan/
E-mail: jpryor@exa.unicen.edu.ar

Abstract. Aspect-Oriented Software Development is an approach which supports the modularisation of crosscutting concerns through the development phases of an application. One of the limitations in the main approaches to AOSD is that they do not provide support for the definition and handling of conflicts that may arise between multiple competing aspects. This paper describes how conflicts are managed by a framework that we have developed for the construction of aspect-oriented applications. In the first place, we describe the main characteristics of this reflective multi-level framework and the tool that instantiates it. We then present our categorisation of different conflicts and how they are defined and handled by our environment for the development of aspect-oriented software.

Keywords Aspect-oriented software development, crosscutting concerns, reflective architectures, aspect conflicts, frameworks.

1. Introduction

Aspect-Oriented Software Development (AOSD) [1] is a widespread and experimented approach to the separation of concerns [2] in software engineering. The goal of aspect-oriented software development is to provide explicit support for modularising the design decisions that overlap or crosscut the functional decomposition of a system. As these features crosscut the primary functionality of the application, their code is spread throughout the basic functional components. In AOSD the crosscutting concerns are encapsulated in separate modules called aspects, whose code is woven into the functional components of the system at predetermined join-points.

Current approaches and techniques for Aspect Oriented Software Development [3][4][5][6] differ on many issues. Some of these issues include the manner and timing for the composition of aspects with other components, whether aspects may be composed with other aspects, how to improve aspect reusability, and how conflicts among multiple competing aspects are solved.

Computational reflection is a technique that permits a system to observe and modify the properties of its own behavior. It is a solution to the problem of creating applications capable of maintaining, using or changing the representation of their own designs [7]. It seems natural therefore, to consider reflection as an adequate technique for the implementation of systems with crosscutting concerns. By its very own definition, the solving of associations between crosscutting and basic concerns may be handled dynamically, as a system observes its behavior and consequently modifies it at runtime.

Our work on reflective architectures in different problem domains including AOSD [8][9] has led to the development of a framework for the construction of aspect-oriented software. This framework was designed to include the characteristics for aspect-oriented applications that we considered would increment the adaptability, flexibility, and reuse of the resulting software.

A tool has also been constructed to facilitate the instantiation of the framework. This tool, called Alpheus, provides an environment which allows the developer to define all the components of the application and their associations, and to visualize them graphically in UML based notation [10]. From these specifications, the tool generates the Java code of the application.

One of the problems of the current approaches and techniques for AOSD is the lack of support for the definition and handling of conflicts between aspects. When more than one aspect is associated to the same object and they are not totally independent, the system´s behaviour may be unpredictable.

In order to explore a solution to these situations in AOSD, we studied the different dependencies that may arise between competing aspects, and defined categories of conflicts. Our objective has been to permit the developer to define different types of conflicts between components of the software, and to include the treatment of conflicts in our environment for AOSD, including the above-mentioned framework and Alpheus, the tool for specifying and generating the application.

The following section presents the multi-level reflective framework. Section 3 describes Alpheus, the AOSD tool. Our categorization of conflicts and how they are defined and handled by the framework and Alpheus, is described in Section 4. Lastly we present our conclusions.

2. A Framework for the Construction of Aspect-Oriented Software

We have developed a framework for the construction of aspect-oriented applications. In the reflective architecture developed to support the administration of aspects or crosscutting concerns, the components of a system reside on two different types of levels:

- The *base level,* which contains the objects that deal with the basic functionality of the application.
- One or several *metalevels,* which contain the aspects or crosscutting components of the application.

The framework, implemented in Java [9], was designed to provide the components and functionality necessary for the construction of quality aspect-oriented applications. It includes *planes* for the grouping of aspects, a variety of reflection and association strategies, and the runtime solving of conflicts between competing aspects.

The framework provides structures called *planes*, where a plane is a set of aspects which carry out a specific functionality, thereby facilitating their reuse.

We have identified and provided different reflection strategies for redirecting the thread of control from the base level to the aspects located in planes at the metalevel, adding flexibility to the reflection process [11].

Conflicts may occur if two or more aspects compete for activation. An object at the base level may be associated to more than one aspect, each with its own behavioural objective. If the task to be carried out by each of these aspects is totally independent of the others, the system will execute without any problem. However, if the competing aspects have some dependency between them, the system will behave unpredictably. The framework supports user definition and runtime solving of different categories of conflicts.

This reflective framework was designed to support the development of aspect-oriented applications, avoiding the tangling of code between functional and non-functional components, and enhancing software properties such as flexibility and reuse.

3. Alpheus: A Tool for Aspect-Oriented Software Development

Alpheus is a visual tool designed to aid users in the development of aspect-oriented applications. The tool supports the following tasks:

- the specification of all the components of an application (planes, basic objects, and aspects), and the specification of the associations and conflicts between the components;

- the visualization of these specifications and components by means of different levels of abstraction and UML based diagrams;
- the generation of the corresponding application code.

The tool allows developers to define the classes of the base application and the aspects using pre-existing ones or creating new ones. A developer may use Alpheus to add aspects to a standard application without modifying its classes, by defining the aspect classes to be added to the application, and by specifying the associations and conflicts between the basic application components and the aspects. For an aspect to be composed (or activated) at runtime with a component of the application (class, object, or method), the corresponding association between that aspect and the reflected component must be specified with Alpheus.

Alpheus also supports the specification of conflicts between aspects. The developer specifies the pair of conflicting aspects (or planes), and then specifies how the conflict has to be solved by identifying the type of conflict (see Section 4).

In order to facilitate the design and specification of an application´s components and their associations and conflicts, Alpheus provides different views of the system. It is possible to identify two main groups of diagrams: firstly, those related to planes, conflicts and association levels; secondly, those related to the UML diagrams [10].

Although this tool for the instantiation of the framework currently generates Java code, it has been designed so that it may be extended in order to generate the application code in other programming languages.

4. Specification and Handling of Conflicts between Aspects

Conflicts may occur if two or more aspects compete for activation. There may be different types of hidden dependencies or conflicts between aspects, and each will require a different solution in order to avoid problems [12][13]. It may be the case that a specific aspect should be executed before others, or that the execution of two aspects may produce an inconsistency that would be avoided if only one of them was executed. In other cases, the corresponding activation of the competing aspects may depend on the current runtime context of the system.

In these and other cases, it would be desirable for the developer to specify the type of conflict between competing aspects, and to describe the actions to be carried out, determining the priorities and activation policy of the conflicting aspects. The reflective framework has been designed to permit the definition and runtime handling of conflicts between competing aspects.

In order to incorporate runtime conflict handling in our framework, we first identified and classified the most common types of conflicts. The conflicts which are detected at runtime are called d*ynamic conflicts*, and the developer must specify what actions are to be undertaken when a conflict arises. It is possible that some situations are not problematic at certain moments of execution, but may be at others. For example, a conflict may be defined between two aspects that are not yet associated to the same base object; thus they do not yet present a conflictive situation, but may do so in the future if they are associated to the same object.

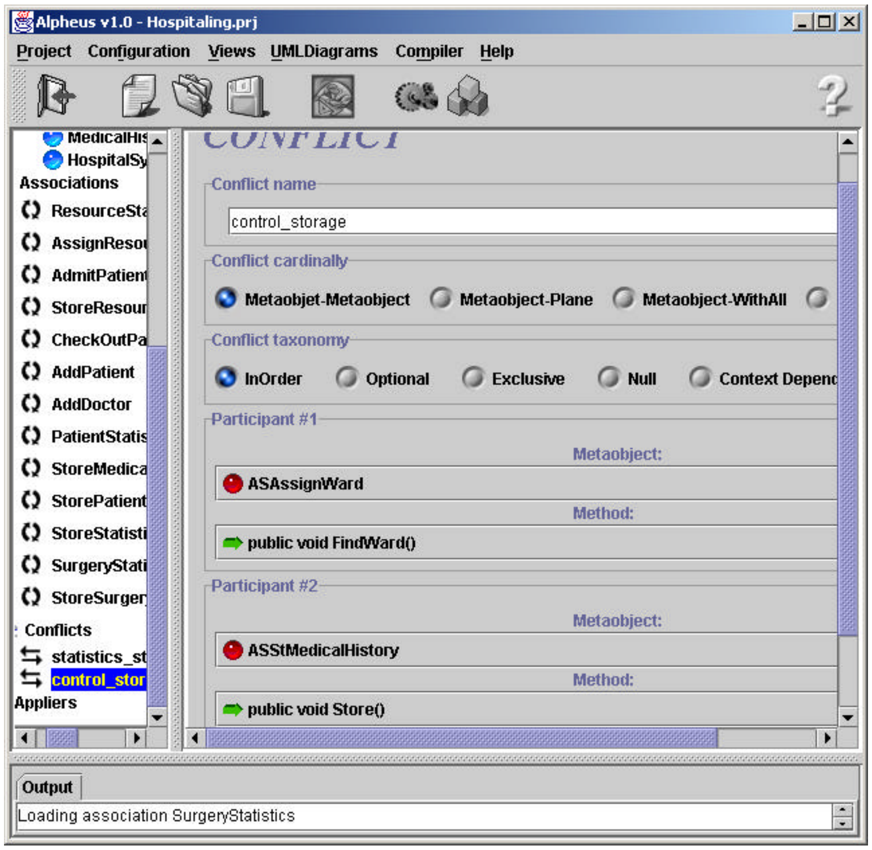
We have identified and implemented different categories of conflict activation policies:

- *InOrder*: the aspects are activated in the order specified by the developer;
- *ReverseOrder*: the aspects are activated in the reverse order to that established by the developer;
- *Optional*: the system itself decides which aspect to activate, according to some pre-established system of priorities or in a random fashion;
- *Exclusive*: only one conflicting aspect is executed;
- *Null*: neither one of the aspects is activated;

- *Context dependent*: the developer adds the code that specifies the activation policy of the aspects.

Although we believe that the above categories of conflicts cover most situations, the framework has been designed to permit the incorporation of other categories. However, this should be unnecessary as context dependent conflicts are coded by the developer and ensure flexibility .

In small systems the declaration and handling of conflicts between specific aspects may be sufficient. However, in more complex and large systems this may become very tedious and difficult to maintain. It is therefore important to abstract the concept of conflicts to a higher level of granularity, permitting their declaration at levels of functionality and not only between specific aspects. With the introduction of planes, the reflective framework supports a more flexible handling of conflicts, easier to define and maintain. The framework supports the following levels of granularity between conflicts: *aspect – aspect*, *aspect – plane*, *plane – plane*, and *aspect - all*.



Conflict name: control_storage

Conflict cardinaly: [✓ Metaobjet-Metaobject] Metaobject-Plane Metaobject-WithAll

Conflict taxonomy: [✓ InOrder] Optional Exclusive Null Context Depenc

Participant #1
Metaobject: ASAssignWard
Method: public void FindWard()

Participant #2
Metaobject: ASStMedicalHistory
Method: public void Store()

Loading association SurgeryStatistics

Figure 1: Specification of a Conflict with Alpheus

Alpheus, the environment which supports Aspect-Oriented Software Development, supports the definition of the different categories and levels of conflicts. Once the corresponding application is generated by Alpheus and it is running, these conflicts are identified at runtime and solved automatically.

Figure 1 shows the definition of a conflict with Alpheus in an example of a Hospital Resource Allocation System. The conflict occurs when a patient arrives and is admitted. The method invoked to add a patient is AdmittoHosp of the HospitalSystem class. This is a reflected method which is associated to the ASAssignWard and ASStMedicalHistory aspects, which assign a ward to the patient and open his medical history. The allocation of resources has to be carried out before the patient´s medical history is opened and updated, so an order conflict is declared between the ASAssignWard and ASStMedicalHistory aspects.

5. Conclusions

This paper briefly describes a multi-level reflective framework and Alpheus, the tool which supports the specification of all the components of an aspect-oriented application and their associations and conflicts. This tool automatically generates the Java code of the application, and also provides the visualization of different UML based diagrams to aid the development process. By using this tool, the resulting aspect-oriented applications are easy to specify and implement. They are also easy to maintain and extend, and their structure enhances reusability. We have also presented our categorization of conflicts between competing aspects, and described how they are specified and handled at runtime.

References

[1] AOSD 2002. 1st. International Conference on Aspect-Oriented Software Development. Enschede. Gregor Kiczales (Ed.). ACM Press. The Netherlands. April 2002.

[2] Dijkstra E.W. A Discipline of Programming. Prentice Hall, 1976.

[3] Aksit, M., Bergmans, L., and Vural, S.: An object-oriented language-database integration model: The composition filters approach. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1992.

[4] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J-M., and Irwin, J.: Aspect-Oriented Programming. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1997.

[5] Tarr, P., Ossher, H., Harrison, W., and Sutton, M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. Proceedings of the International Conference on Software Engineering (ICSE), May 1999.

[6] Pawlak, R., Seinturier, L., Duchien, L., Florin, G. JAC: A Flexible Framework for AOP in Java. Reflection'01 The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns Kyoto, Japan. September 25-28, 2001, Kyoto, Japan.

[7] Maes, P.: Concepts and Experiments in Computational Reflection. In N.K. Meyrowitz, pages 147-155.

[8] Pryor, J., and Bastán, N.: A Reflective Architecture for the Support of Aspect-Oriented Programming in Smalltalk. Position paper at the Workshop on Aspect Oriented Programming of the European Conference on Object-Oriented Programming (ECOOP), 1999.

[9] Valentino, F., Ramos, A., Marcos, C., and Pryor, J.: A Framework for the Development of Multi-Level Reflective Applications. In Proceedings of the Second Argentine Symposium on Software Engineering (ASSE). Argentina, 2001.

[10] Booch, G., Rumbaugh, J., and Jacobson, I. The Unified Modeling Language. User Guide. Addison-Wesley, 1999.

[11] Marcos, C.: Patrones de Diseño como Entidades de Primera Clase. PhD. Degree Dissertation. Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN), Facultad de Ciencias Exactas, ISISTAN Research Institute, April 2001.

[12] Nuseibeh, B., Kramer, J. and Finkelstein, A. A Framework for Expressing the Relationships between Multiple Views in Requirements Specification. In IEEE Transactions on Software Engineering, October 1994.

[13] Truyen, E., Vanhate, B., Joosen, W., Verbaeten, P., Nørregaard Jørgensen, "Dynamic and Selective Combination of Extensions in Component-based Applications", in Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001), May 2001, Toronto, Canada.