Modeling Aspects using Software Stability and UML

M.E. Fayad
Computer Engineering Department
San Jose State University
One Washington Square
San Jose, CA 95192-0180
Ph: 408-924-7364
Fax: 408-924-4153
Email: m.fayad@sjsu.edu

Anita Ranganath
Computer Engineering Department
San Jose State University
One Washington Square
San Jose, CA 95192-0180

Email: anitaarun@hotmail.com

ABSTRACT

Aspect oriented design is an emerging paradigm in software development. It can be thought of as a counterpart to object-oriented design that can aid in supporting orthogonal abstractions – concerns that tend to cut across various components, or which tend to affect more than one class. UML is a standard methodology[11] adopted for object-oriented design. This paper looks at the characteristics of aspects that need representation and the possible UML elements that can be used to achieve this. Briefly, we also describe how the software stability modeling concepts can be useful in identifying aspects.

1. INTRODUCTION

An aspect is a modular unit that encapsulates a crosscutting concern[8,9]. Aspects and components together model the concerns in a system. Components are modular units of functionality whereas an aspect is an abstraction of a concern that crosscuts several components. The components are identified based on an object-oriented approach. This area has undergone considerable research and can be represented well using UML notation. After proper design with UML notation, the implementation phase becomes better laid out. With aspects however, it is not the same case. They have no standard way of being represented in UML models and moreover, aspects at the present time do not have a clear way of being linked to UML models.

As a result, the structure of aspect implementations and their relationship with other UML model elements is unclear. This makes aspect modeling is target language dependent. Once a model is created, it becomes hard to re-use because of the lack of standardization in representation. This calls for defining ways of representing aspects. The following have to be kept in mind - one needs to build UML models to represent the structure of aspects. There must be a way to mark all model elements affected by aspects. Aspects could affect different classes and also be affected by different classes.

Aspects may affect classes at particular points in the class's structure that could be referred to as attachment points. Hence attachment elements must be defined in UML. Aspects are dynamic in nature and hence may affect different classes at different points in time; so the order must be shown when multiple aspects are being applied. Mechanisms must be provided for expressing both the points at which elements of different concerns must be joined and the relationships that govern how the joins are made.

2. ASPECTS IN THE DESIGN PHASE

Aspects usually are considered to play an important role in the implementation phase. However, the basis for a good implementation lies in a good design and analysis phase. In this phase if the aspects in the system being modeled are identified and if the classes that they will affect are determined, then the implementation becomes much simpler. The Software Stability modeling concepts introduced in [1,2,3] can be a very useful way of identifying aspects. Using this approach, we find the core goals of the system defined by the EBTs (Enduring Business Themes), and the workhorses of achieving these goals called the BOs (Business Objects) and then the application specific classes called the IOs (Industrial Objects) that could be replaced without causing a rippling effect through the entire model. The EBTs capture the core goals of the system. The BOs make the EBTs which are implicit concepts proposing the ways to achieve the concepts. The architecture of the software stability models is therefore layered in nature making it easy to view the classes that would be affecting other classes or components (refer Figure 1). Most EBTs and BOs in the stability model cut across other classes and hence could be modeled as aspects. The Stability model makes it convenient to understand the classes that would have a lasting effect on other classes (that is the EBTs) since they form the innermost layer and the BOs that form the next layer. It will be advantageous to model the BOs as aspects because they affect two layers of classes in the model as they lie in between. They have an effect on the EBTs and the IOs of the system. If they are modeled as aspects, then they are given room to change over time and become better maintainable. If aspects are introduced in the design phase, then it may be possible to view the situations where particular aspects recur and this can lead to the development of patterns in aspect oriented development which can be very useful as it will promote reuse. Also, modeling can serve as good documentation to provide a better understanding of the problem considered.

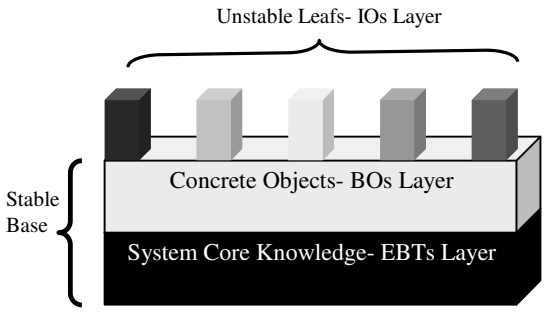

Figure 1. Software Stability concepts layout

Let us consider a simple e-commerce example as an illustration. A customer makes purchases online using a credit card. The customer can view and select items which he can add to his shopping cart. The shopping cart will help the customer purchase a number of items in a single transaction. The class diagram using the traditional modeling approach is as shown below in Figure 2.

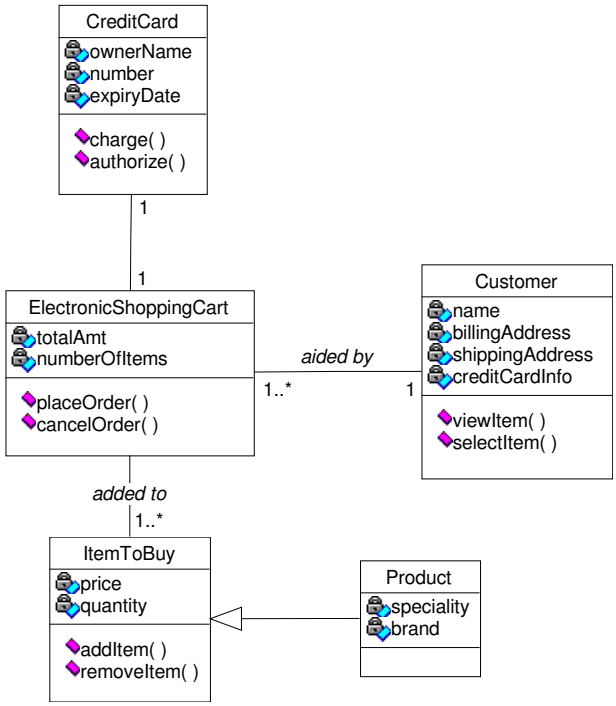

Figure (2) – Traditional Model (E-commerce example)

The same problem when modeled using the software stability concepts yields the following solution.

In the stability model (Figure 3), we recognize the classes Trading, Convenience and Security as the EBTs since they form the core goals in this example. EBTs are conceptual and hence we need the more concrete BOs to achieve these goals. For example, the EBT of Security is achieved through an Authorization class that encapsulates the authorization procedure methods.

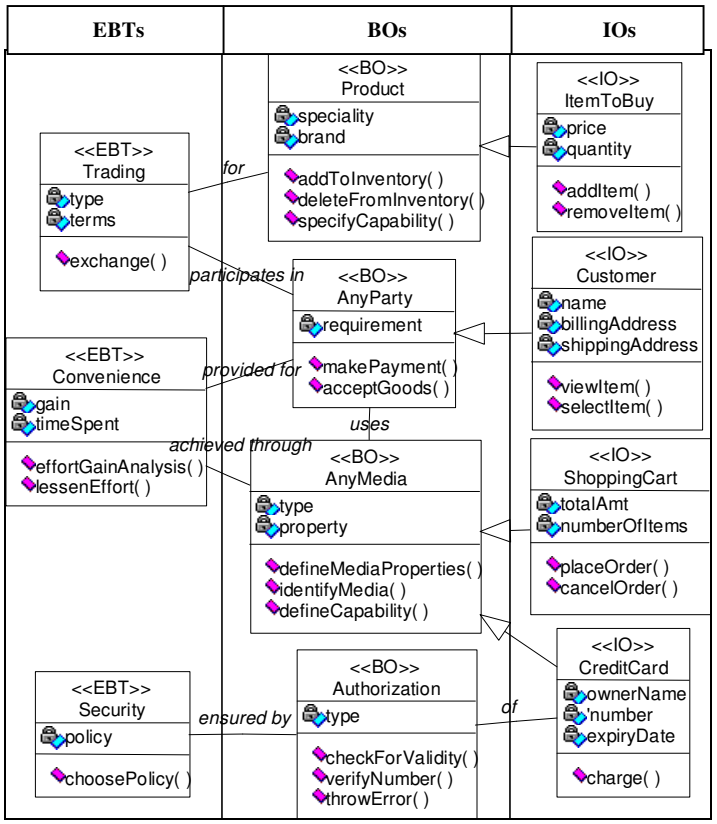

Figure (3) – Stability model for E-commerce example

Then at the periphery, we use IOs that are application specific. From the above model, it is easy to see that Security (which is in the innermost layer of EBTs) has an effect on the classes (Authorization, CreditCard) in the layers above it. So security has an affect on one or more classes and hence can be modeled as an aspect. The modeling using the software stability concepts helped us recognize that security was a concern that needed representation, which was not the case in the traditional model. Moreover, the authorization class cleanly encapsulates the needed procedure for security in the example.

Designs of most systems are aimed at achieving maximum efficiency and performance. These will be modeled as EBTs in the stability model. The conceptual goals are recognized through the BOs of the system. The model provides consideration of the goals right from the modeling stage. For example, instead of looking at persistence, performance, and quality-control as additional features, one can model them as EBTs and clearly define the classes that work towards them (through BOs).

The EBTs and BOs can be implemented as aspects since they have an affect on one or more classes. Moreover, they can evolve independently and as the code is now localized and hence easily maintainable. Using the software stability concepts may make the recognition of aspects easier when compared to the traditional modeling approach. It will also streamline the process of aspect-oriented development.

However, there maybe cases where aspects are needed temporarily. Take an example where you use a logging aspect for debugging. The logging code is needed only during testing, and once the code is working as desired, the logging code can be removed. In such cases, though the logging aspect does have a prevailing presence during the testing phase, it may not be required after this stage. Modeling the logging code as an aspect can be helpful in inserting it uniformly in all the classes as well as removing it uniformly once the task of debugging is completed. In such cases, where aspects just make a task easier but have no core functionality in the system, we do not represented them as an EBT or BO in the stability model. However, these aspects are easier to recognize since they are needed only in the development stage and they involve a routine insertion of code without really affecting the functionality of the code.

3. PROPERTIES OF ASPECTS THAT NEED REPRESENTATION

Consider this example. Assume there are classes called 'Circle', 'Square', and 'Rectangle' (refer to Figure 4). Each of them has methods that define how to draw the particular shape in its particular 'draw' method. In addition, the class will have properties required to define that shape – for example the circle will have a center defined by a point and a point and a radius whereas a rectangle will have four points indicating its vertices. Assume the purpose is to display each of the above shapes. So any time the draw method is called, one needs to call the display method. The display method is now common to all the three shape classes. The code for the display method will have to repeat in each of the shape classes – it is said to crosscut these classes. Moreover if we add another shape to be displayed, then the display method will have to repeat in it too. It can be said that the display method crosscuts the shape classes. If we model the display method as a separate entity (called aspect) that can be used by each of the classes, then we have isolated the code for displaying, avoiding unnecessary repetition of code. This also makes the easily maintainable because all the changes have to made in one place.

Let us first look at the terms that go with aspect-oriented programming[4,6,7] with respect to the example above. These are with reference to the programming constructs used with AspectJ[5].

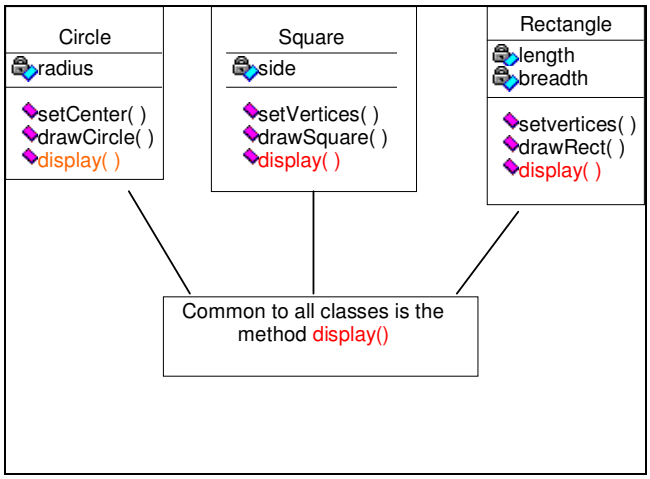

Figure (4) – Example of Aspect

3.1. Aspect as a module: Aspect is a modular unit and may be treated like a class in some sense in that it implements a feature's behavior and declares when that feature must execute just as a class in object oriented programming encapsulates data and functions to manipulate the data. These will resemble the attributes and operations or methods of classes. However, the difference between aspects and classes in object-oriented programming is that an aspect represents a common concern that crosscuts other components whereas a component is usually designed to handle one specific functional concern. In the illustration above, displaying is an aspect since it has affect in each of the shape classes and it does not show some unique behavior that is characteristic of the class it is present in. The display code is essentially the same in each of the shape classes.

3.2 Join Point: These are the points where one or more classes crosscut. It could be at a particular method call, a constructor call, read/write access to a field, exception handler execution, object and class initialization execution. Join points will be clear points of execution within a program. In the example above, the drawCircle(), drawSquare(), drawRect() are all joinpoints. After each of them occur, we need a call to display(), the aspect comes into play when these methods occur.

As another example consider a case where the execution of a particular function or a call to a particular method needs to be logged. This function or method will be part of a particular class or interface. Instead of including the log procedure within the class where the method is present, one could define a join point at that particular method and indicate a link to the log procedure. So the method becomes a join point. We must provide for a way to represent these join points. These must be marked and associations must be made to the aspects that will affect them.

3.3 Pointcut: An aspect affects many classes. The points in the classes where the aspects functionality is used are the joinpoints. There could be a collection of join points indicating all the classes affected and the points where the aspect's functionality comes into play. This is referred to as a point cut. Point cuts are essentially a set of joinpoints. Also, the point cut defines exactly the position in the class where the aspect applies. For example, it could be before the method in the class, after the method, or in particular condition. For the case in Figure (2), the point cut will be the collection of the three join points - drawCircle(), drawSquare(), and drawRect(). The point cuts could be viewed as pure weaving instructions. One should be able to indicate all the joinpoints and the operation (called advice) that needs to be executed and where.

3.4. Advice: An advice specifies what is to be executed upon reaching a join point. It could be a method call, an object instantiation, a field access, etc. It could also be a control where the execution of the method or any other operation depends on certain conditions. The advice could execute either before, after, or around a join point's execution. A 'before advice' will execute before the join point; an 'after advice' will execute after a join point. An 'around advice' acts like a control, indicative of whether the particular method or function at that join point should execute or not. The above example illustrates the case of an after advice. It is a requirement that after the shapes have been drawn, the display is updated. So the advice in this case will be the method of displaying.

As an example of the around advice, consider the case where one needs to apply synchronization. The classes which participate define a method **lock()** and a method **unlock()** which must be called before and after the actual work is done (called semaphores). Here the lock and unlock procedures can be encapsulated in a synchronization aspect and in this case will be like a control with respect to those classes.

3.5 Introduction: An introduction is used when one needs to add more capability to an existing set of classes by adding new class members like constructors, methods and fields. Sometimes the classes may be part of an inheritance tree. With the help of an introduction, one can add capability to a set of such classes thus altering the inheritance relationship. When an introduction is used for a non-derived class, it merely enhances the class's capability. The introduction is weaved in statically at compile time.

In addition to the above elements, one needs a representation for aspect priority in a multi-aspect environment. When a class has multiple aspects affecting it, there has to be an indication of order of application of the aspects.

So, basically we need to represent two kinds of crosscutting – one is *structural* and the other is *behavioral*. An example of structural crosscutting is the introduction. Behavioral crosscutting is expressed through the join points, point cuts and advices.

Aspects can be represented as stereotypes of standard UML classes [11]. It can have attributes and operations, and may represent a subclass of an existing modeling element (class), but has a different implication.

Structural crosscutting, such as introductions, can be represented using collaboration templates. Behavioral crosscutting forms a part of UML interaction diagrams and is shown at the *link* that is used to communicate the message between objects. Weaving instructions for introductions define the classes in the base hierarchy that need to be crosscut, that is, the actual arguments to the collaboration template parameters. On the other hand, for advices, weaving instructions specify which links in a base model need to be crosscut – the points at which the advices need to be run.

The join points are distinct points of execution in a program. So they are shown highlighted in UML interaction diagrams. Pointcuts imply where the advices act and can be represented as stereotypes within an aspect structure. Advices are operations, always associated with a point cut, and are also part of the aspect structure. The figure shows a representation for aspects.

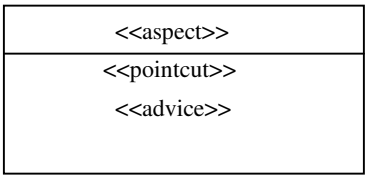

Figure (5) - Representation of an aspect

4. SOFTWARE STABILITY AND AOP

We looked at aspects and software stability model in Section 2. The EBTs and BOs that form aspects are identified in the design phase. They are then mapped to AOP concepts discussed in Section 3 and represented in UML. This forms the basis for the implementation phase of the software application. Thus, starting from Software Stability concepts in the design phase ensures stability in the system while incorporation of aspects renders the system adaptable and easily maintainable.

CONCLUSION

The advantages of aspect-oriented programming are many. Aspect oriented design can play a very important role in aspect-oriented programming approach since it can help in identifying aspects at an early stage. However, a good modeling notation is necessary. Allowing UML to support aspect-oriented design can be very advantageous as UML is widely adopted and understood.

The usefulness of software stability concepts in identifying aspects is illustrated with an example. Starting from a stable model for the system, the EBTs and BOs are implemented as

aspects. The principles of AOP and software stability work to the advantage of each other both ensuring the stability of the system being modeled and its maintainability.

REFERENCES:

[1] M.E. Fayad, and A. Altman. "Introduction to Software Stability." Communications of the ACM, Vol. 44, No. 9, September 2001.

[2]M.E Fayad. "Accomplishing Software Stability." Communications of the ACM, Vol. 45, No. 1, January 2002.

[3]M.E. Fayad. "How to Deal with Software Stability." Communications of the ACM, Vol. 45, No. 4, April 2002.

[4] Gregor Kiczales, John Lamping, Anurang Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irving. Aspect Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka (Eds.), Proceedings of ECOOP'97. Jyvaskyla, Finland. June 3-13, 1997. Lecture Notes in Computer Science. Vol. 1241. Springer-Verlag, pp. 220-242.

[5] AspectJ website www.eclipse.org

[6] I want my AOP! Parts 1, 2, 3 in Java World by Ramnivas Laddad

[7] Aspect Oriented Programming, by Tzilla Elrad, Robert. E. Filman and Atef Bader from Communications of the ACM, October 2000 Vol 44.

[8] Discussing Aspects of AOP by Tzilla Elrad from Communications of the ACM, Oct. 2001

[9] Aspect Oriented Programming, Gregor Kiczales et al, Proceedings of ECOOP, 1997

[10] Unified Modeling Language User's Guide, Grady Booch, James Rumbaugh, L.Jacobson

[11] Designing Aspect Oriented Crosscutting in UML, Dominik Stein, Stefan Hannenberg and Rainer Unland, Aspect Oriented Modeling with UML, (as part of the AOSD), 2002.