Use Cases as Aspects – An Approach to Software Composition

Dean Wampler, Ph.D.
Aspect Programming, Inc.
dean@aspectprogramming.com

Draft V0.1

ABSTRACT

Aspect-Oriented Software Development addresses the "cross-cutting concerns" that compromise the modularity of Object-Oriented systems by modularizing these concerns as *aspects* and by providing mechanisms to combine aspect and object modules to compose applications. Use Cases are also cross-cutting concerns. This paper discusses a vision of Domain-Driven Development where use-case-driven software development is treated as a form of AOSD. Use cases are implemented as independent modules and applications are composed by combining use case modules. The potential advantages and disadvantages of this approach are discussed, along with the issues that require further investigation and the expected evolution of this approach as AOSD techniques emerge and evolve.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming.

General Terms

Design, Languages, Theory.

Keywords

Aspect-oriented programming, object-oriented programming, use cases, software development process.

1. INTRODUCTION

Aspect-Oriented Software Development [6] is a new approach to software development that addresses limitations in Object-Oriented Software Development, namely the loss of system modularity in the face of "concerns" that cut across the boundaries of the main object model decomposition, which is the dominant form of modularity used in OO applications. For example, security is a common concern that requires consistent handling across a collaborating set of objects, in some, but not necessarily all circumstances.

AOSD offers a new form of modularity designed to encapsulate these cross-cutting concerns and thereby restore overall system modularity. These concerns or *aspects* are identified, modularized, developed independently, and then combined with the main object model[1] in a structured way to compose the application, a process called *weaving*.

[1] In fact, AOSD is not limited to object-oriented systems. [2]

Use-case modeling is a traditional form of analysis that focuses on how the system will be used by people or other systems to accomplish particular goals [1]. The user-centric view of the system helps to prioritize lists of features and requirements and helps to ensure that the resulting system meets usability needs, *etc*. Hence, use-case modeling and requirements elucidation should be done in tandem, since they feed and refine each other. Since the use case model and requirements specification reflect the problem domain of interest and because they drive the rest of the development process, use-case-driven development is a form of domain-driven development.

In fact, Ivar Jacobson has recently demonstrated that a use case satisfies the definition of an aspect [3, 4]. A typical use case is realized by a collaboration of objects and each object may contribute to multiple use cases. Hence, use cases are cross-cutting concerns, with respect to the object model. Therefore, the principles of AOSD offer new insights and techniques for use-case-driven development, which we explore here.

2. WHY TREAT USE CASES AS ASPECTS?

2.1 Problems in Use-Case Modeling Today

In traditional use-case modeling [1], the analyst defines a list of activities and services that the system will provide to external *actors,* which are external systems or *roles* performed by people. Use cases provide a very high-level, black-box view of the system and they are typically described textually, with a few use-case diagrams to show relationships between them. The information gleaned from this modeling exercise drives the list of requirements and features, as well as their priorities. The requirements and features, in turn, drive the list of use cases and their definitions. It is an iterative and incremental process.

Once the use cases have been defined, the next step is to model how they are *realized* by objects. Typically, this involves identifying the central objects in the system and describing how they interact with one another, using sequence diagrams, activity diagrams, *etc.* This model is the analysis model, if it remains at a level of abstraction above any implementation details, or a design model, if it is allowed to incorporate architectural and other design details. For our purposes, we will assume that an analysis model is constructed.

There are two big challenges in the realization process. The first is the manual nature of mapping between the descriptions of the use cases and the analysis model that realizes them. The analyst goes through the use case descriptions and manually identifies potential objects that participate in the realization. Then the behaviors and the collaborations between those objects are identified. This manual process is slow, tedious,

Copyright © 2003 Dean Wampler, All Rights Reserved

and error prone, requiring process discipline and careful checking that all the use cases are properly mapped to the analysis model and to ensure that no superfluous analysis elements have been introduced. Clearly, automation of this mapping process would be helpful.

The second challenge is that all the use cases are considered at once and one object model is built for all of them. This makes it difficult to partition the work among a team and it imposes the difficult task of juggling a lot of information at once. This further complicates the manual process.

These problems only get worse as the implementation proceeds, since the amount of detail increases as you go down to lower levels of abstraction.

Jacobson's AOSD approach discussed here addresses the second problem by partitioning the larger analysis model and implementation into separate models and implementations, one model and implementation for each use case [3]. These use case *modules* [4] are combined together to compose applications. The complexity problem is reduced using "divide and conquer". This approach addresses the first problem indirectly; while it does not define a means of automating the mapping from use cases to object models, it makes the manual process simpler due to the better management of system complexity.

2.2 Implementing Use Cases, One at a Time

AOSD offers a new form of program modularity and composition tools that help fulfill the long-term vision of developing applications through composition of components. The observation that a use case loosely fits the definition of an aspect suggests that a potential solution to the problems identified previously is to implement each use case separately, in isolation, and then to construct the application by weaving the resulting use case implementations together.

For example, Team 1 implements Use Case Modules A and B, Team 2 implements Use Case Module C, and Team 3 implements Use Case Modules D and E. The application is then composed by weaving together the separate implementations. In principle, this weaving could also be done at a higher level of abstraction, where one application model is created and it is used to generate the implementation. Here, we assume that weaving occurs at the implementation level. We will discuss alternatives later.

The first complication that comes to mind is that redundancies and gratuitous differences will arise, especially when different teams implement different use cases. For example, different names may emerge for the same conceptual element or the same name may be used for *different* concepts. In principle, the weaving process can resolve these problems by merging redundancies and providing methods to rectify naming and other discrepancies. However, the prevalence of these problems can be reduced with a few simple practices.

The development process proceeds as follows:

1. Identify use cases in the usual manner [1].
2. Capture a glossary of terms and concepts. Refer to them frequently to minimize gratuitous redundancies (see below).
3. Establish conventions for naming, common idioms, *etc.* (again to reduce gratuitous redundancies).
4. Implement each use case separately.
    a. Use the glossary and the conventions to identify, name, and realize concepts in a consistent manner.
    b. Communicate regularly with other teams to identify and remove emerging redundancies and gratuitous differences.
    c. Implement the use case.
5. Assemble the application by weaving. (See the next section)

To begin, use case identification proceeds normally. [1, 10]

Each use case is realized using an object model defined for it alone and the implementation for this use case is generated from the object model [4]. Only the essential classes, attributes and methods needed to implement the use case are created, even though it is expected that other use cases may require conceptually-similar classes with additional methods and attributes. The weaving process will handle the merging of these classes.

Notice that by focusing on just what is needed to satisfy the use case, unnecessary characteristics, even if "expected" for future needs, are not implemented, thereby avoiding wasted effort. This supports one of the best practices of Agile Programming [6]; only implement what is necessary.

A glossary of concepts that arises from the use cases and requirements provides the "vocabulary" for the application. Combined with the development conventions, separate teams working on different use cases are more likely to use the same terms and constructs, if they share a common "vocabulary", thereby minimizing gratuitous differences and making application composition easier later.

Frequent communications between teams also help reduce these problems. However, the teams should avoid the temptation to "economize" and reduce redundancies too much at this stage, because doing so can compromise the modularity the process is designed to create. On the other hand, it is appropriate, given the schedule and resource constraints faced on real projects, for teams to do frequent *refactorings* to extract common, reusable information. The term *refactoring* refers to the process of performing structural modifications that preserve existing behavior but set the stage for future evolution of the system.

These points suggest that very frequent iterations with small increments of change should be used. The weaving process, discussed below, is used to compose the application in the current iteration. Then the use case modules that were woven together (not the resulting application) are refactored to extract common information and to set the stage for the next iteration of development. Note that because the refactorings are behavior preserving, if the weaving is repeated, it should yield an application with identical behavior.

This process reduces the complexity of realizing use cases, because the realizations are decoupled into smaller modules, one for each use case. Hence, the amount of information and complexity that must be managed is reduced through "divide and conquer".

3. ASPECT WEAVING FOR APPLICATION COMPOSITION

The biggest challenge facing this approach is how to create robust and automated aspect weaving tools and processes with

Copyright © 2003 Dean Wampler, All Rights Reserved

which applications can be composed. Automation is essential. Without it, this approach is too tedious and error prone to be viable. AOSD tools and processes are evolving rapidly. The most mature tools are source-code oriented tools, especially for Java. Hence, weaving at the implementation level is the most viable today. We consider two tools, AspectJ and Hyper/J (CME).

3.1 AspectJ

AspectJ [7] is the most popular and mature tool for doing AOSD in Java. It uses Java language extensions to define aspects and the *pointcuts* into which they are woven. A pointcut is a set of well-defined points in the application where additional code can be inserted. These points are called *join points* and they include entry and exit points for methods, variable modifications, *etc*. AspectJ's pointcut language is a powerful and expressive means of specifying very broad or very specific pointcuts into which aspect code is to be woven, either at the source code or the byte code level.

However, AspectJ has one conceptual bias that limits its usefulness for our purposes. That bias is an orientation towards aspects as relatively small modules that are woven into larger object structures. This bias is reflected in the pointcut language and the conventional ways in which aspects are implemented in AspectJ. In contrast, the bias of this paper is towards large "sibling" aspect structures, one for each use case, which are combined together. Inside the aspect structures, you will find object structures.

3.2 Hyper/J and CME

Hyper/J [9] and its emerging successor, the Constraint Manipulation Environment (CME) [8], are projects at IBM Research resulting from years of research into ideas that extend OOSD, including AOSD. While less popular with Java developers, Hyper/J has a number of conceptual ideas that make it more suitable for program composition through aspect weaving, as envisioned here [3]. However, Hyper/J is limited to Java byte code. Hyper/J's successor, CME, will extend these capabilities to arbitrary source languages and binary formats, as well as higher-order abstractions and representations, such as UML.

Hyper/J implements the idea of *multi-dimensional separation of concerns* (MDSOC), where each concern (aspect) occupies a dimension in a hyperspace. This concept is used to support both decomposition of existing classes into separate concerns (dimensions), as well as the composition of classes from separate concerns. A rich control language provides the directives to Hyper/J for performing these actions.

For example, suppose that an implementation for one use case, *Start System,* contains an *Administrator* class and the implementation for another use case, *Manage Users,* contains a *Manager* class, but they represent the same concept, so they should be combined. Each class implements a set of methods and attributes needed for its particular use case. The sets may or may not overlap. Using Hyper/J's composition language, the application composer specifies that these two classes will be merged, with the resulting class taking the name *Administrator,* and the methods and attributes in each class will be added to *Administrator*. Among other things, the control language can also specify how to assign override priorities when conflicts are encountered, how to combine method bodies, *etc.*

Hence, today it is possible for Java applications to be implemented through composition of separate use-case implementations, at the Java byte code level, using Hyper/J. As CME evolves, it will be possible to do this weaving at the source code level in Java and other languages, as well as at higher-levels of abstraction, through weaving of UML models.

4. PROGRAM EVOLUTION

For this approach to be viable, it is also necessary for it to support typical, real-world concerns faced by development teams. An important issue is requirements change. Changes can occur during the course of a single application release cycle and they certainly occur between cycles. These changes fall roughly into two broad categories, (i) modifications to the existing structure and (ii) additions that are structure preserving. New use cases could fall into either category.

Modifications to the existing structure will require refactoring and new development, as discussed previously. In contrast, an addition can often be isolated in a new aspect and woven into the existing application in a way that is transparent to the older use cases.

The use case *extension* mechanism fits the addition situation nicely, where the change aspect is implemented as an *extension* use case. The *extended* use cases can remain agnostic to the change.

The structure changing use case is likely to be a "non-extension" use case, which we refer to as a *primary* use case, because it is usually more fundamental to the application and affects the structure of the composed application more deeply.

5. OPEN ISSUES AND FUTURE WORK

5.1 Application Architecture

Nontrivial applications should have well defined architectures. How should the architecture be developed in this approach?

The architecture will likely emerge two ways. First, some up-front, business, platform, and tool decisions will often mandate a particular *framework*, such as J2EE or .NET for enterprise applications. Second, the application-specific architecture will emerge during the implementation, through refactoring and iterative development.

However, the architecture itself is also a cross-cutting concern, since it, too, spans object boundaries. Hence, some architectural elements should be implemented as aspects and woven into the use case implementations.

Best practices for architecture development will need to be developed as experience is gained with this approach.

5.2 Appropriate Levels of Abstraction for Weaving and Refactoring

Weaving and refactoring at higher levels of abstraction can be simpler than working at the implementation level, because of the reduced amounts of detail, especially if modeling tools support the require manipulations graphically. Yet, this approach can also have dramatic effects, because the higher-level abstractions drive lower-level details. This is the vision of OMG's Model-Driven Architecture initiative [4]. However, the most appropriate levels for weaving and refactoring will

Copyright © 2003 Dean Wampler, All Rights Reserved

depend on emerging tools and best practices, gained through experience.

In the near term, CME will offer the best tooling for application composition through use-case weaving at the analysis and design levels, because of its planned support for UML/XMI and EMF. CME has a strong model of weaving "program slices" together to construct programs.

Today, composition must be done at the source-code or byte-code level using Hyper/J or AspectJ. As discussed previously, Hyper/J has better constructs for application composition as discussed here.

5.3 Practical Experience

This vision for use-case-driven development using AOSD needs to be tested and refined through use in real, non-trivial projects. This effort will also reveal best practices and drive the evolution of modeling and weaving tools.

Note that this approach to use case realization and program composition is consistent with principles of Agile Programming [5] and RUP [10], which emphasize implementing only just what is required to satisfy the stakeholder requirements today, using refactoring to evolve the architecture, and iterative development cycles. Hence, this approach encourages several best practices for software development.

6. CONCLUSIONS

The observation that use cases fit the definition of aspects opens up new possibilities for use-case-driven development using AOSD techniques. In particular, we have discussed how use cases can be developed independently of each other, as separate aspects, and then woven together to compose applications.

Note that early work on AOSD focused on the cross-cutting concerns that affect the dominant decomposition, the object model. In a sense, the concerns are "small things" affecting the dominant decomposition or structure of the application. The logical progression of this idea is for the application structure to be a composition of aspects, each of which is composed of objects and smaller aspects. The idea of use cases as aspects reflects this shift in thinking about structure.

How realistic is this approach? Clearly AOSD is relatively new and immature. To date, there have been no projects that attempt the approach described here. Hence, these ideas need to be tested and refined.

Existing modeling tools already permit modeling of use cases and requirements. Some can also drive the generation of artifacts at lower levels of abstraction. However, modeling of aspects and weaving is in its infancy.

Robust and capable weaving tools are necessary at all levels of abstraction where weaving is to be performed. AspectJ and Hyper/J can be used for weaving implementation artifacts, with limitations. Hence, automated weaving of use cases can be done at the implementation level today. CME, when mature, will support weaving at higher levels of abstractions.

7. ACKNOWLEDGMENTS

Thanks to Ivar Jacobson, Magnus Christerson, and Grady Booch for helpful discussions.

8. REFERENCES

[1] Bittner, K. and Spence, I., Use Case Modeling, Addison-Wesley, Boston MA, 2003

[2] Coady, Y. and Kiczales, G., "Back to the future: a retroactive study of aspect evolution in operating system code", in Proceedings of AOSD 2003 (Boston MA, March 2003), ACM Press, 50-59.

[3] Jacobson, Ivar, "Use Cases and Aspects - Working Seamlessly Together", invited talk, AOSD 2003 (Boston MA, March 2003).

[4] Jacobson, Ivar, "Use Cases and Aspects - Working Seamlessly Together", in Journal of Object Technology, Vol. 2, No. 4, July-August 2003, pp. 7-28.

[5] Wampler, D., "The Role of Aspect-Oriented Programming in OMG's Model-Driven Architecture", *in press*. (Draft available at http://www.aspectprogramming.com/)

[6] The Agile Alliance web site. http://www.agilealliance.com/home.

[7] The Aspect-Oriented Software Development web site. http://aosd.net/.

[8] The AspectJ web site. http://www.eclipse.org/aspectj/.

[9] The Concern Manipulation Environment web site. http://www.research.ibm.com/cme/.

[10] The Hyper/J web site. http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm.

[11] The Rational Unified Process web site. http://www.rational.com/products/rup/faq.jsp.

Copyright © 2003 Dean Wampler, All Rights Reserved