RC22409 (W0204-096) April 24, 2002
Computer Science

IBM Research Report

A Position On Considerations In UML Design of Aspects

William H. Harrison, Peri L. Tarr, Harold L. Ossher
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

A Position On Considerations In UML Design of Aspects

William Harrison (harrisn@watson.ibm.com), Peri Tarr (tarr@watson.ibm.com), Harold Ossher (ossher@watson.ibm.com)

Abstract

Aspect-Oriented Software Development has been popularized primarily as an approach to coding since the late 1990's [9, 10] although put forward as an approach applicable to the larger design-through-coding aspects of development since the early part of that decade [1, 7]. In addressing the need for design representations of the same kind of separation of concerns addressed in coding, a rush to simply represent coding-level artifacts in the design must be avoided, and the issues of difference in intent and difference in expression of various design artifacts must be taken into account. This paper addresses some of the considerations.

The following sections discuss the general impact of level-of-design, of packaging, and of aspect attachment, and then review several relevant UML diagrams to discuss the impact of these on how aspect information may best fit into those diagrams.

Level of Design

UML [3] is used both for high-level design or architecture of systems and for low-level design of their implementation. In defining ways to represent the elements of separated concerns in UML, it is important that the elements fit the level of the design. The level of design is important to consider because, depending on the level, different expectations can be placed on the details of the artifacts that aspects contain. In turn, the expectations about artifacts influence how the aspects or concerns containing them can be related and combined.

Low-Level Design/High-Level Code

The low-level design of a system can also be viewed as the high-level coding of its implementation. Some part of the implementation code can be generated directly from the design without imposing particular restrictions or conventions or making significant implementation choices for the developer. The class diagrams in a low-level design intended to be mapped to Java, like that in Figure 1, would be expected to contain both classes and interfaces, in their appropriate roles. It would be expected to reflect Java's inability to support multiple inheritance among classes. It may even be expected to contain adornments, stereotypes, or tagged values that specify class or association details like persistence, serialization, or peculiar concepts of scope or visibility. Many UML tools and environments [11-17] provide automated mapping from low-level design to implementation.

High-Level Design

The high-level design can also be viewed as an architectural statement, one that is intended to umbrella many different implementations. The entities in a class diagram, like that in Figure 2, represent the real or conceptual things being manipulated, without implementation distinctions like interface, abstract class, or instantiable class and with a generalization structure that reflects the classification of things rather than how implementations are shared. It is still possible to provide some automated generation of code from high-level designs, but more decisions are taken from the developer into the hands of the generator's designer. Tools like Tengger [8] and methodologies like Catalysis [6], and more domain specific model-driven generators [5] like that from SoftMetaWare [16], provide for model-driven generation from high-level designs.

Packaging

A second issue in reflecting aspect elements in a UML design is their packaging (in the broader sense of the word). While UML provides a packaging construct, the conventional uses of its packages reflect an implementation packaging. If packaging constructs are to be used to encapsulate higher order organization of packages, aspects for example, additional capabilities will be needed. For example, UML's packages are related only by dependency relationships. The use of packaging constructs with the need for association-like relationships is illustrated in the UML design approaches taken in [4].

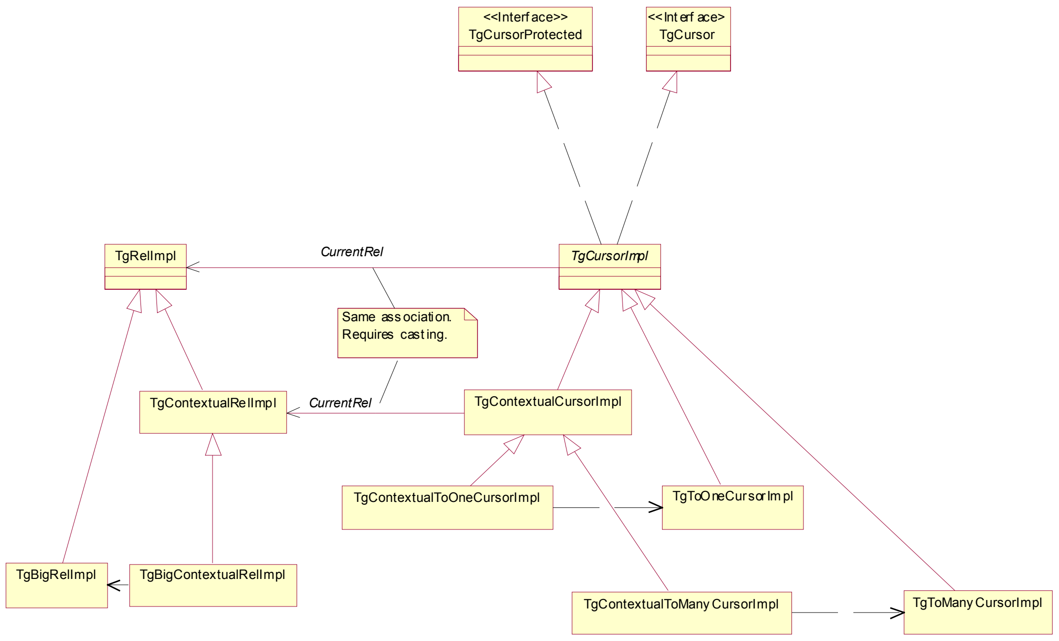


Figure 1 – A Low-Level Design's Class Diagram

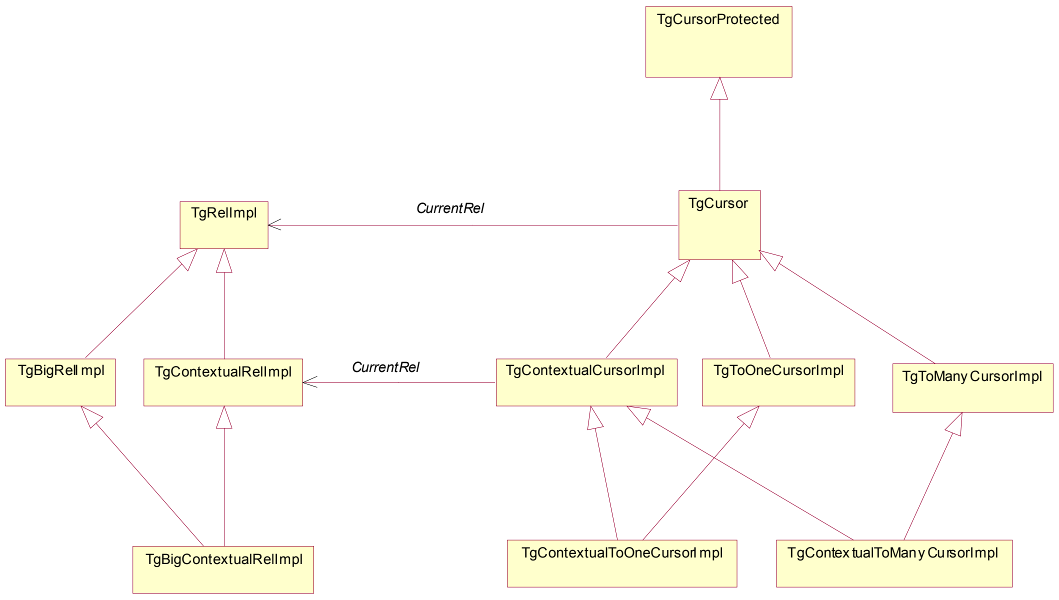


Figure 2 – A High-Level Design's Class Diagram

The introduction of UML support for aspect packaging can provide the ability to track the collection of additions to a design that occur as the design is refined. To facilitate high-level design with refinement downwards, one need to be able to depict the change in artifact interrelationships over time or in response to requirements, In figure 3, for example, class X is described rather incompletely at one level of design with only its association to class Y, with no details provided yet. When refinement of the design to address some requirement takes place, the refinement is encapsulated in a concern that shows the original class refined into two subclasses, only one of which has the association. The "refinement" association suggests replacement of the original definition rather than simple merging.

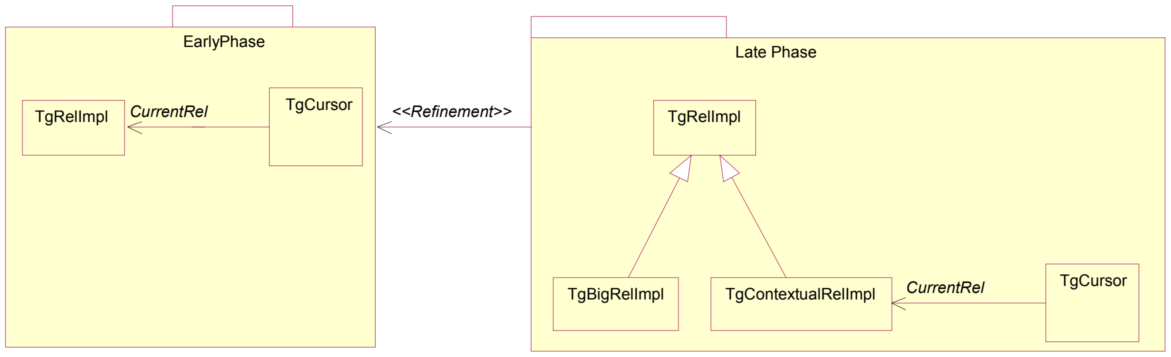


Figure 3 – Using Aspects For Refinement

Aspect Attachment

In addition to providing attachment elements in a concern-separated UML design, mechanisms must be provided for expressing both the points at which the elements of different concerns must be joined and the relationships[1] that govern how the joins are made.

Code-Based Attachment

Certain artifactual elements are present only in code. The most evident one is the sequence of references to state variables, to built-in (extra-design) elements like the language's built-in and library operations. While a design's interaction diagrams may constrain or describe the sequence of reference to model elements, good high-level designs generally avoid overspecification of the sequence. Artifactual elements like line numbers, call points, execution traces, etc. are available for code-based attachment.

However, while most forms of advisory and other joining relationships apply equally well to both code-based and design-based attachment, one of them, "around", is worthy of special note and may require special handling in design situations. "Around" advice presumes two conditions, illustrated in Figure 4: 1) there is a clearly identifiable base body of material to be wrapped around and 2) the material wrapping the base body contains a special annotation of where the base body is to be employed. The first of these conditions causes aspect-aspect interactions that require higher-order advisement to complete the semantics. The second of these requires language extensions or conventional extensions be employed and that the concern containing the aspect is unusable as a stand-alone concern. In the simplest of cases, where the execution of the base simply falls somewhere in the middle of the aspect's code, the code would better have been written with before/after advice. When that is not possible, design-level diagrams generally do not carry so much detail that the attachment point can be described.

Design-Based Attachment

While suitable for injecting debugging, tracing, and other such aspects into an existing implementation, code-based attachment does not form a good basis for aspect-oriented design because its attachment points are at too low a level. At a higher level than code (even at the level of a low-level design) classes, operations, attributes, associations and interactions are the primary elements that are available for attachment of aspect logic.

But most forms of joining relationship that can be used at for code-based attachment are also applicable in design-based attachment as well. This includes before, after, before/after, as well as various combination functions for results. Figure 3 illustrated the use of a "refinement" joining relationship, a form of override.

[1] These are called "advice" in AspectJ, but advice is too easy to ignore to use the term in its more general setting.

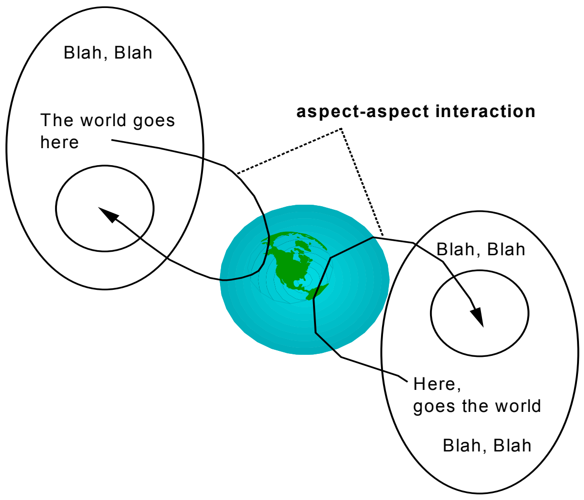


Figure 4 – “Around” Wrappering Advice

Diagram Families

UML Diagrams fall generally into two groups. Structural diagrams, like class diagrams, component diagrams, deployment diagrams, or state machines, characterize the static structure and relationships of the design elements. Behavioral diagrams, such as use-case diagrams, interaction diagrams, sequence diagrams, collaboration diagrams, activity diagrams, work-flow diagrams describe or constrain the interaction and flow of information and control among those elements.

Structural

Time and space for the moment limit discussion to class diagrams and concern diagrams.

Class Diagrams

Class diagrams carry the design for the objects that contain logic and state. The introduction of aspect-oriented design approaches can be expected to impact these diagrams in the following ways:

Linguistic classifier marking: Some UML classifiers may be stereotyped to indicate that they are to be coded in special ways that depend on their role in relationships that describe the joining of join-points. Strictly speaking, such stereotyping, like an “aspect” stereotype, is of the same nature as stereotyping that describes or constrains the implementation language or style. This sort of stereotyping ought be unnecessary since the same information is conveyed by the presence and nature of packaging and join specifications.

Join-Relationship Marking: The relationships that specify how the elements are to be brought together are another natural point of extension. The relationship is naturally in the nature of an association, and stereotyping or the indication of association classes can be used to indicate the form of the advice, whether in the form of a simple before or after, or in the more complex forms indicated in [4]. While not advocating this form, Figure 5 illustrates how an aspect containing an aspect, depicted as a “beforeToday()” operation, can be shown to apply to all “remove()” and “setOtherEnd()” methods in a package might be depicted as a modified association.

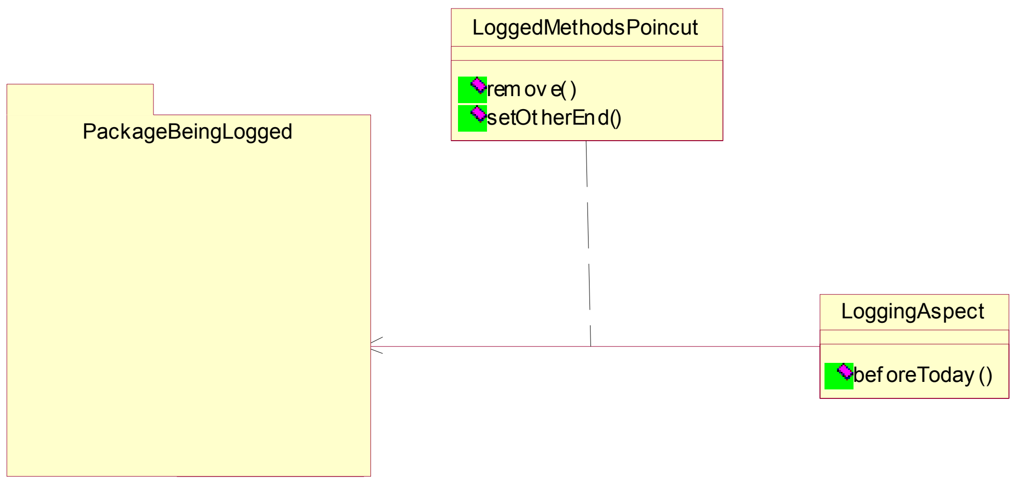


Figure 5 – A simple concern diagram for advising placement of an aspect

<u>Join-Point Annotation:</u> While join points for method entry/exit and for all call-points with some predicated qualification in a class can be inferred from the ends of structural join-relationships, other characterizations, such as particular call points in the program flow are better annotated in behavioral diagrams.

Concern Diagrams

Concern diagrams are not yet part of UML, but any discussion of extending UML to deal with aspects needs to address them. Broadly, concern diagrams are an extension of UML's component diagrams. Current usage of component diagrams deals mostly with the low-level design of software. Concern diagrams can be used at both high and low levels of design.

Concerns are classifiers with features that describe their content and their expectations. Features, like packages, can contain classes and relationships among them. But the features themselves can be associated in ways like that shown in Figure 4. In addition to the associations that describe their composition relationships, e.g. "advice" information, concerns can be related by dependencies or generalizations. The associations and dependencies between concerns can carry constraints and other common annotations.

Behavioral

Time and space for the moment limit discussion to interaction diagrams. There are two primary interaction diagrams in UML: sequence diagrams and collaboration diagrams. The two are technically interchangeable, but they expose different information for easy visualization

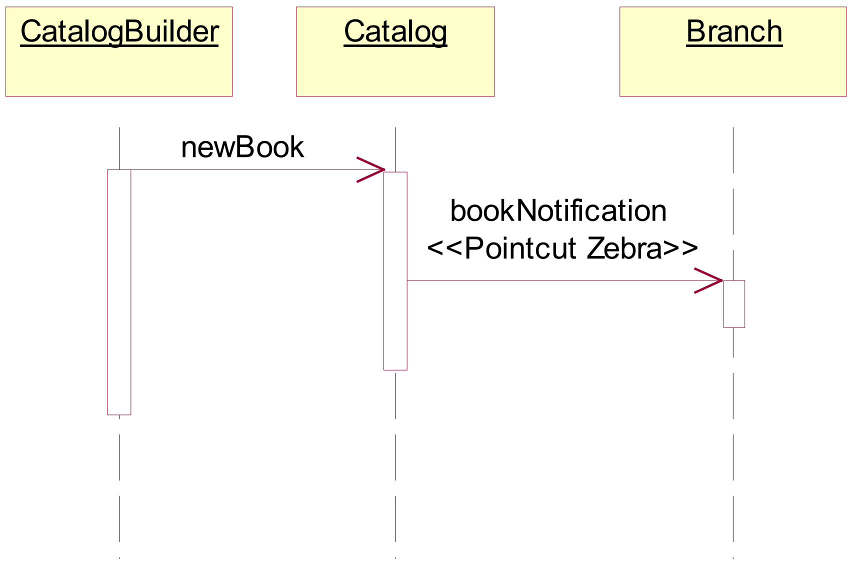
Sequence Diagrams

<u>Join-Point Annotation:</u> Sequence diagrams expose more information about the behavior of an object, showing, in time-sequence, a series of interactions among objects. It is reasonable to use the interactions as identifiers of particular call-points within a static or dynamic cascade of activity. For example, in a particular sequence it may be possible to speak of "the calls from A on method X of B in response to a call on a call on method Y of A".

However, sequence diagrams have considerable ambiguity that must be dealt with in pursuing this course. For example, the first call indicated may not actually refer to the first call, but only to some call, or to all calls in the first loop. Other sequence diagrams may indicate different courses of action under other circumstances. Similarly, the first call out in execution might arise as one of several possible first calls guarded by conditionals in the code.

Without descending to the code itself, some tightening of the meaning of sequence diagrams will probably be necessary to use interactions as join-point annotations. This tightening will need to be accomplished in a way that does not require code and, preferably, in a way that even applies to high-level designs.

Join-Relationship Marking: Having identified an interaction as a join-point of interest, it must be connected to a join-relationship. The naïve approach would be to try to use an association between the interaction and the aspect to be activated. But this will likely be unwieldy, requiring extra object-lifelines for the aspects, and un-UML-like, requiring associations between relationships (the interactions) and lifelines. A more promising approach might be to use the *pointcut* concept described in [10]. The interaction could be annotated with the names of pointcuts to which it belongs, as illustrated by Figure 6.



Collaboration Diagrams

Collaboration diagrams embody similar information to sequence diagrams, but present that information differently. Collaboration diagrams effectively collapse all the interactions between any given pair of objects into a single interaction. This makes specifying a pointcut that contains all of these interactions easier, but does not resolve any of the other issues that were discussed in connection with sequence diagrams.

References

[1] Aksit, M., Bergmans L., Vural, S. “An object-oriented language-database integration model: The composition filters approach.” In Proceedings ECOOP'92

[2] Andersen, E.P., and Reenskaug, T. System design by composing structures of interacting objects. In O. Lehrmann Madsen, editor, ECOOP '92: European Conference on Object-Oriented Programming, pages 133-152, Utrecht, June/July 1992. Springer-Verlag. Lecture Notes in Computer Science, no. 615.

[3] Booch, G., Rumbaugh, J., and Jacobson, I., “The Unified Modeling Language User Guide,” Addison Wesley (1999)

[4] Clarke, S., Harrison, W., Ossher, H., and Tarr, P., “Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code,” *Proceedings of the 1999 Conference on Object-Oriented Programming Systems, Languages, and Applications,* Denver (November, 1999).

[5] Czarnecki, K., and Eisenecker, U. Generative Programming: Methods, Techniques, and Applications. Addison-Wesley, 2000

[6] D'Souza, D., and Wills, A. C. " Objects, Components, and Frameworks with UML – The Catalysis Approach," Addison-Wesley (1998)

[7] Harrison, W., and Ossher, H. "Subject-oriented programming (a critique of pure objects)." In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA), September 1993.

[8] Harrison, W., Barton, C., Raghavachari, M. Mapping UML Designs to Java, Proceedings of 2000 Conference on Object-Oriented Programming Systems, Languages, and Applications, Minneapolis 2000

[9] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes C., Loingtier J., Irwin J. "*Aspect-Oriented Programming."* In Proceedings ECOOP'97, Springer-Verlag, June 1997.

[10] Kiczales, G., Hilsdale, E., Hugunin J., Kersten, M., Palm, J., Griswold, W. *"An Overview of AspectJ."* In Proceedings ECOOP'01, Springer-Verlag, June 2001.

[11] –, Aonix, Software through Pictures (MetaEdit+), http://www.metacase.com/

[12] –,AppBuilder, http://www.devdaily.com/AppBuilder/.

[13] –,No Magic, Magicdraw, http://www.magicdraw.com .

[14] –,Object International Software, Together/J, http://www.togethersoft.com/downloads/index.jsp.

[15] –, Rational Software, Rational Rose, http://www.rational.com/products/rose/index.jtmpl.

[16] –, SoftMetaWare, http://www.softmetaware.com/services.html

[17] –, Softera, SoftModeler, http://www.softera.com/manual/UserGuide.htm.