Representing Aspects in the Software Architecture – Practical Considerations

Paulo Merson
Software Engineering Institute
Pittsburgh, PA 15213-3890, USA
pfm@sei.cmu.edu

ABSTRACT
Languages, tools, libraries, as well as success stories from an active community are available today to programmers that decide to adopt AOP. However, little support is available to architects who want to capture in the software architecture the elements and relations from AOP. This paper provides some guidance to representing aspects in the architecture, building on approaches suggested by others for modeling aspect-oriented software with UML extensions. Aspects are first-class elements in the module view of the architecture. The UML language can be extended to represent aspect modules, along with their special attributes: named pointcuts and advices. Different authors have suggested the explicit representation of the crosscut relation between an aspect and other modules. The representation of this relation in the architecture is not practical in general. This and other considerations related to architecture documentation are discussed in the paper, using elements drawn from the Java Pet Store application.

Keywords
AOP, AOSD, software architecture, modeling, UML.

1. INTRODUCTION
There are plenty of resources available in terms of languages, tools and aspect libraries that realize at the coding level the paradigm introduced by aspect-oriented programming (AOP). An active community of developers has been exchanging information and reporting success stories of AOP implementations in projects that span various business segments and development platforms. At the architecture level, equivalent concrete resources and results are not available. The architect willing to embrace aspect-oriented software development will find little guidance with respect to efficient techniques to capture in the architecture the elements and relations introduced by aspect orientation.

The goal of this paper is to introduce a pragmatic approach to the documentation of software architectures for systems built with AOP. Different approaches making use of the Unified Modeling Language (UML) have been proposed to represent the design of aspect-oriented software. This paper evaluates benefits and difficulties these approaches may exhibit when applied to software architecture representation. Alternatives that promote feasibility and simplicity are suggested.

Section 2 presents a basic notion of view-based architecture documentation. Section 3 describes how aspects could be documented in the software architecture, contrasting different UML-based approaches found in the literature. Section 4 uses the Java Pet Store application as an example to consolidate ideas suggested in the paper. Finally, Section 5 provides some concluding remarks and insights on directions for future work.

2. ARCHITECTURE VIEWS
Software architecture is the structure or structures of a system, which comprise the software elements, their externally visible properties, and the relationships among them [1]. The term "structures" in this definition can be interpreted as architecture views. Thus, the architecture of a software system consists of multiple views. These views vary in nature. Some views show the organization of the code units (e.g., packages, layers, classes, programs) and are the blueprints for implementation. Others show an "x-ray" of the system when it executes, exposing elements that have runtime presence (e.g., processes, threads, EJBs, data stores). Yet other architecture views are important, such as one that explains how the system is deployed to communicating processing nodes (deployment view).

Each architecture view defines the types of elements and relations that can be represented in that view. For example, in a module view we may find boxes that represent classes or packages, but should not see a box that corresponds to a printer, which belongs in the deployment view.

2.1 Module View of the Architecture
The architecture view that shows the structure of the software in terms of code units is called module view. Elements in this view are generically called modules, which represent different types of code units depending on the target implementation platform. For example, modules in a Java application are typically packages, classes and interfaces, whereas in C they are programs and possibly header files. Three types of relations are commonly seen in module views [2]:

- A *is part of* B: containment relation. For example, a Java class is part of a package.
- A *depends on* B: dependency relation. For example, if a Java class A has calls to methods in class B, there is a usage dependency (shown in UML with the <<use>> stereotype).
- A *is a* B: specialization/generalization relation. This relation is typical of object-oriented languages. It is used when A is a subclass of B, or when B is an interface and A implements B.

Aspect-oriented programming complements object-oriented programming by providing a means to modularize cross-cutting concerns. The next section discusses how aspects in the implementation should be represented in the module view of software architecture.

3. ASPECTS AS MODULES

Consider an application to be developed using AspectJ. At the architectural level, the module decomposition will typically show Java packages, classes and interfaces. The architecture should also show aspects, because they become code units in the final implementation. An aspect is a special type of module in the module view and should be differentiated from classes and other types of modules in the notation. A class module has attributes and methods. An aspect module, in addition, has (named) pointcuts and advices[1]. Aspect modules have a special type of relation with other modules. When a pointcut is used in an advice, it defines a relation between the aspect module that encloses the advice and the modules that have join points matching the pointcut. This relation is referred to as "crosscut" to indicate that the aspect module crosscuts the other modules.

The language used to represent the module view of the architecture should provide a notation to represent aspect modules, their pointcuts and advices. The representation of crosscut relations between aspect modules and other modules in the architecture does not seem to be essential for reasons that are discussed later in this paper. Next we will look at UML notation to represent aspects in the module view of the architecture.

3.1 Representing Aspects in UML

UML is a visual language to specify, construct and document the artifacts of a software system. UML is a natural choice for most architects because it became a *de facto* and *de jure* standard for software design representation. Among the 13 different types of diagrams available in UML 2.0 [3], class diagrams and package diagrams are particularly suitable to describe the static structure of a system in terms of implementation units. In effect, UML has been widely used to represent the module view of the architecture of object-oriented systems.

UML provides extension mechanisms that can be employed to create additions to its metamodel. *Stereotypes* can add (specialize) types of elements or relations, and *tagged values* define properties of stereotyped elements and relations. Stereotypes and tagged values may be packaged together in profiles.

Different authors have used UML to represent the design of systems that contain aspect modules [4,5,6,7,8,9]; even a UML profile for aspect-oriented software development has been proposed [10]. The common denominator among these UML representations is an <<aspect>> stereotype applied to the class or classifier metamodel element. Denoting aspect modules as stereotyped classes undoubtedly makes sense, since aspects are structurally similar to classes—aspects may contain attributes and operations, and may extend another aspect in inheritance relationships. The different approaches used in the literature to represent aspects in UML sometimes lack in practicality when it comes to represent the crosscut relation and the special attributes of aspect modules (pointcuts and advices).

[1] Using static AOP, aspects may also include inter-type and compile time declarations. This paper focuses on advices; a discussion of static AOP mechanisms is left for future work.

3.1.1 Crosscut Relation

A stereotyped UML association or relationship with the <<crosscut>> label has been suggested to represent the relation between an aspect and the modules that are crosscut by that aspect [5,10]. Similar approaches also proposed include: a stereotyped relationship called <<binding>> [7], a stereotyped association called <<pointcut>> [8], and a UML association with tagged values [9]. The <<crosscut>> stereotyped relation and its variations present a practical issue. The notation requires drawing a line (with the <<crosscut>> label) from the aspect module to all other modules that contain matching join points (Figure 1). In many cases, too many lines may be necessary and the diagram can easily get cluttered. Moreover, every time a new class that contains matching join points is added somewhere in the design, we have to draw a line from the aspect module to that class. Updating the diagram becomes cumbersome, unless the design tool can automatically identify the crosscutting relation. Even if such tool existed, it would not work in all cases because often times the details of the new class are not specified in the design to the degree needed to match join points.`

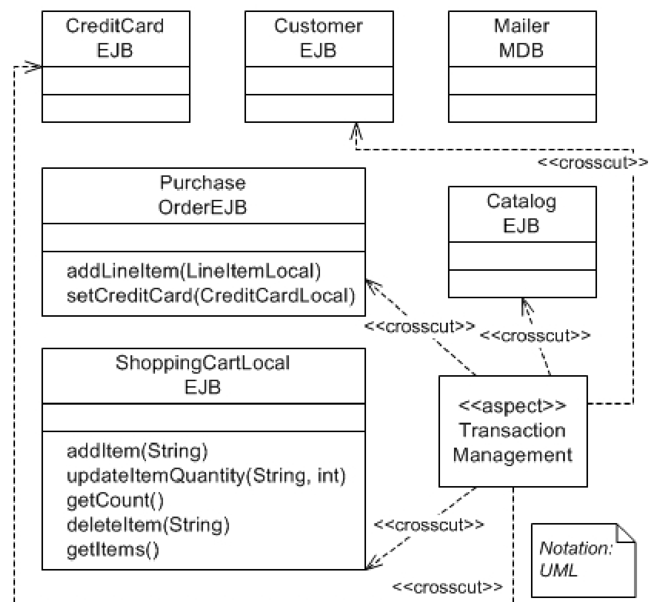

Figure 1. Crosscut relation in a UML class diagram.

To avoid tangling in design diagrams, simply do not draw lines for crosscut relations. A better alternative is to follow the same simple representation used in AOP. When we create a pointcut specification in an aspect-oriented language, we do not have to indicate all classes that contain the target join points. Pointcuts are specified using signature patterns with wildcards or metadata annotations. The same thing at the design level should suffice.

If the lines representing the <<crosscut>> relations are removed, one may argue that the diagram becomes less expressive. An equivalent argument applies at the programming level though. Moving crosscutting logic to aspects yields classes that are easier to read exactly because they contain core logic only (no tangling). On the other hand, when we look at base code we no longer see the crosscutting logic that will be woven. In some situations (e.g., maintenance), not knowing what strange things advices might be doing behind the scenes negatively affects modifiability. This is a

modifiability vs. modifiability tradeoff that is inherent to AOP. However, the benefits of reduced tangling and scattering far exceed apparent difficulties related to obliviousness. Besides, AOP IDEs can indicate in the base code where a join point is matching the pointcut of an advice. For instance, the code editor in AJDT (www.eclipse.org/ajdt) displays gutter annotations with graphical icons that tell the programmer if a before, after or around advice applies to that line. Hopefully one day an equivalent mechanism will be available in an automated design tools—diagrams could use icons to designate the presence of an advice (Figure 2) and hovering the mouse over an icon would show a tooltip informing what aspects crosscut that element. Until then, the architecture will more likely not show crosscut relations.

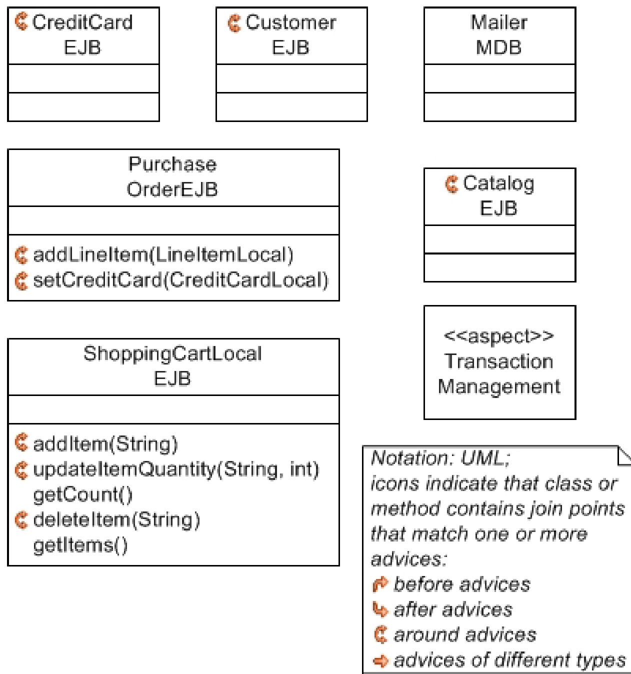

Figure 2. Icons instead of lines to indicate crosscut relation.

3.1.2 Pointcuts and Advices

Aspects contain attributes, operations, pointcuts and advices. Assuming the stereotyped <<aspect>> element is a subclass of the UML class model element, it is natural to use the attribute compartment for the aspect member variables and the operation compartment for the aspect methods. To list named pointcuts and advices, the most appealing alternative is to create additional compartments. The UML specifications are quite flexible with respect to classifier compartments [3]: any compartment can be added; in diagrams, compartments can be suppressed for readability; compartment names can be used to remove ambiguity. Thus, <<aspect>> modules should have two extra compartments, one named "Pointcuts" and the other "Advices" (Figure 3(a)).

The major benefit of having separate compartments is that pointcuts and advices can be easily recognized. However, many UML tools will not support additional compartments. An alternative is to list pointcuts and advices in the attribute or operation compartments using stereotypes. Let's look at advices first. An advice is structurally similar to an operation in the sense that it can take arguments, may return a value (around advice) and has a body of instructions that is executed when the advice is called. Therefore, it makes sense to list advices in the operation compartment with a stereotype to distinguish them from methods. The use of different stereotypes for each kind of advice (<<before>>, <<after>> and <<around>>) has been proposed [8,9]. The problem is that at design time we may envision an advice, but we may not know yet whether it will be best implemented as a before, after or around advice—analogously, we sometimes envision a method in a class at design time but we can't yet determine if it will have public, protected or private visibility. Furthermore, different AOP platforms support distinct kinds of advices that would require more stereotypes (e.g., <<after throwing>>, <<after returning>>). The proliferation of specialized stereotypes may hinder the generality of the design with respect to implementation. A more practical alternative is to have a generic stereotype <<advice>> applied to operations. This alternative has been suggested in the aspect-oriented design model (AODM) approach [5]. The authors of AODM also prescribe the creation of pseudo identifiers for advices to overcome a limitation of UML that does not allow two operations in the same classifier to have the same signature. The current version of the UML specifications do not impose that operations have unique names. In fact, the specifications state that the parameter list, which may be the only thing differentiating overloaded operations, can be suppressed [3]. In any case, the signature of an <<advice>> operation is different from the signature of a UML operation, so an option is simply to omit the operation name in <<advice>> operations. It makes sense to give names to <<advice>> operations for documentation purposes or in case the target AOP platform supports advice names though. Otherwise, advice specification in the design should be kept simpler than it is in the implementation.

In the signature of the <<advice>> operation, we *optionally* add the word "after," "before" or "around" to designate the kind of advice. The advice signature should also have a pointcut specification that may or may not use named pointcuts declared in that aspect. Figure 3*(b)* shows an aspect that contains an operation with the <<advice>> stereotype.

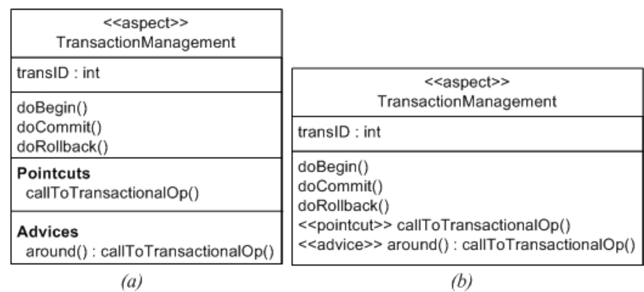

Figure 3. Aspect with pointcut and advice.

As we did for advices, if we cannot add a compartment to list named pointcuts, we can specify them as either stereotyped attributes or operations. There is no strong reason to opt for one or the other. Using the operation compartment for pointcuts is the choice in AODM [5]. It seems to be the best option because the proximity of stereotyped pointcuts and stereotyped advices may improve readability, and because pointcuts can take arguments

just like an operation. Figure 3*(b)* shows a stereotyped <<pointcut>> in the operations compartment.

A formal syntax for pointcut specification is necessary when the design is input to code generation. A design tool that could create icons as in Figure 2 would also require a formal syntax. However, the syntax used in pointcuts is a potential source of confusion in design diagrams. Renaud Pawlak and associates suggest a variety of symbols and keywords combined with regular expressions for identifiers [8]: *!* denotes method invocation; *?* denotes method execution; *<N>* denotes instance creation, *ALL* points to all the methods of a given class, and so on. Ivar Jacobson and Pan-Wei Ng specify pointcuts in a separate compartment using a different syntax [6]. In their notation, for example, "roomAccessOp = *(..)" corresponds to the following pointcut definition in AspectJ (ReserveRoomHandler comes from a different compartment called "class extensions"):

```
Pointcut roomAccessOp() :
  withincode(void ReserveRoomHandler.*(..));
```

There are three options to write clear pointcut specifications in the design. One is to stick with the syntax used in the implementation platform (e.g., AspectJ) or a subset of it. Creating a distinct syntax for design just increases the burden on developers. Another option is to use natural language in comments or annotations that would let the reader of the architecture documentation to understand what join points is the pointcut targeting. Although natural language is inherently ambiguous, many times it is the best architects can use without spending too much time in the design. The other option is simply to omit the binding expression in the design. In this case, the architect should use a name for the pointcut that is intuitive and be available to explain the pointcut to the reader. Figure 3 specifies a pointcut named "callToTransactionalOp()" but the pointcut specification is relegated to the implementation phase or later in the design.

A different approach that has been suggested is to specify pointcuts as separate elements in the design [7,9], instead of properties of the aspect module. The stereotyped <<pointcut>> becomes the part instance in a whole-part relationship where the whole instance is the <<aspect>>. However, there are reasons not to make pointcuts first-class elements in the module view of the architecture. Pointcuts are very important to understand how aspects crosscut the base classes, but, like in the implementation, they are just pieces of aspects. Additionally, several named pointcuts can be defined in an aspect. Creating boxes for each pointcut around the aspect could clutter the diagram. It would be harder to read an advice specified inside the aspect because it may refer to several pointcuts surrounding the aspect.

3.2 Aspects in Other Architecture Views

The previous sections describe how aspects can be specified in the module view of the architecture using UML. So far we have not discussed the representation of aspects in other architecture views. Aspects are implementation units that do not correspond to any runtime component because they stop existing as units after the weaving process. For example, the runtime view of the architecture of a J2EE application shows servlets, EJBs, and many Java objects. Even if that application was created with AspectJ, the runtime view will not show the aspects individually, because their logic is now dispersed in the binary code of the runtime components. Therefore, runtime views of the architecture do not change due to the use of aspects in the implementation. The deployment and implementation views [2], which show allocation of software elements to hardware elements and file system elements respectively, also remain unchanged by the presence of aspects in the implementation.

Therefore, information about aspects in the architecture documentation is to be found basically in module views. The next question is whether a module view using a notation different than UML would help to convey architectural information. The answer is yes due to at least one example. The AOP community has used a "bars and stripes" informal notation to represent the weaving of aspects (denoted as colored horizontal stripes) across multiple classes (denoted as vertical bars). This graphical representation has been automated in the Visualiser Eclipse plug-in (www.eclipse.org/ajdt/visualiser). Figure 4 shows a screen snapshot. The caveat is that it is not feasible to manually create and update these diagrams. However, when the aspect-oriented code already exists, it shows in a succinct and neat notation the base modules and how do the aspects crosscut them.

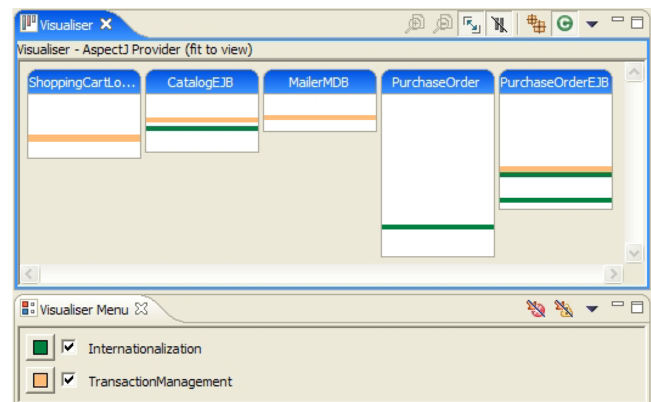

Figure 4. Bar and stripes notation shows crosscutting advices.

4. EXAMPLE: PET STORE

The Java Pet Store application is a J2EE application created by the Java BluePrints program from Sun Microsystems (http://java.sun.com/blueprints). The original implementation available for download does not contain aspect modules. Figure 5 shows the high level module view of the architecture reconstructed from the source code and deployment descriptors. The top level packages with the <<app>> stereotype correspond to the four applications deployed independently: the *admin* rich client application, the web-based *petstore* application that allows customers to browse and buy pets, the *opc* application that has server-side components to process orders asynchronously, and the web-based *supplier* application that controls the inventory of the pet supplier. Figure 5 is the high-level module view; the internal structure of each package is depicted in separate (module) views in the complete software architecture document.

There are many crosscutting concerns in the Java Pet Store that could be implemented with AOP. One of them is transaction demarcation. The original implementation uses the transaction support provided by the J2EE platform. An AOP-based solution could give more flexibility to the developer in terms of the actions to take in each step of the transaction lifecycle and in terms of using datasources that do not provide a transaction interface.

Another crosscutting concern is internationalization. In the web-interface, the customer can click on different flags to select a different idiom. That action represents a *locale* change. Locale is also set during deployment. Changing locale results not only in changing the idiom of all messages displayed to the user, but also the currency used for payment, the text in email notifications, and the catalog of pets that the user can browse. An aspect can help to modularize the actions that should be taken when the locale is set. Figure 6 shows these two aspects in UML using the notation suggested in Section 3. Note that in the TransactionManagement aspect, the pointcut specification is just a note in natural language, whereas in the Internationalization aspect, the pointcut is specified using AspectJ syntax. The crosscut relation from these aspects to the other modules is not represented in the architecture, as suggested in Section 3. Intuitively, one can imagine that these aspects crosscut many modules in different packages—it is not worth trying to draw lines for all these relations.

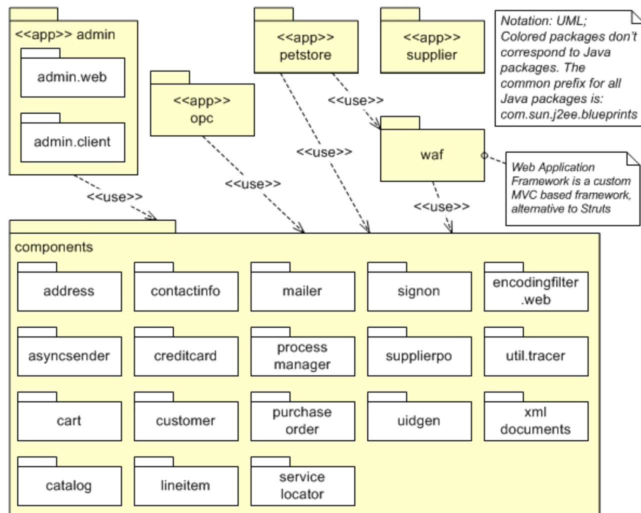

Figure 5. High level module view of the Pet Store application.

Figure 6 does not show in what package the aspects reside. The general question is whether *all* aspects should be collocated in a separate package. In a large application, there can be some pervasive aspects such as TransactionManagement and Internationalization that could be placed inside an isolated package for aspects. Nonetheless, there can be application specific aspects that are used in confined parts of the application. For example, the Pet Store could have an aspect that performs shopping cart repricing upon operations such as quantity updates and delivery option changes—a similar aspect was implemented for a commercial J2EE application [11]. The best place to put the cart repricing aspect is probably the cart package (under the components top level package). The architect should not put all aspects in a separate package only because they are aspects, just like he or she should not put all interfaces in a separate package or all abstract classes in a separate package.

5. CONCLUSION AND FUTURE WORK

The module view of the architecture of a regular Java application does not show every single class and interface used in the implementation. Some classes and interfaces are simply not relevant at the architectural level. Likewise, not all aspects created in the implementation will be architectural, although the crosscutting nature of aspects may make them more likely to be deemed architectural.

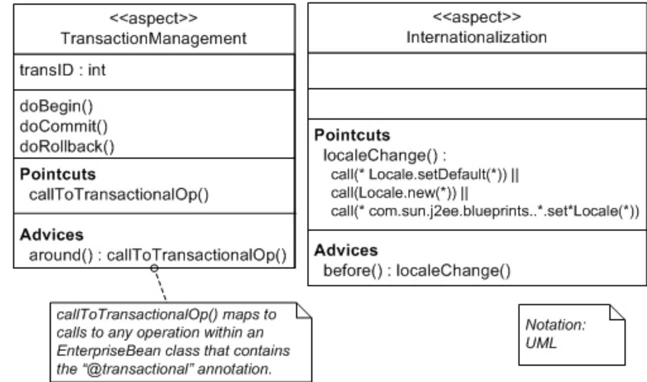

Figure 6. Two candidate aspects for the Pet Store application.

In architectural diagrams, it is not feasible to have as much detail as we do in the code. The representation of pointcuts and advices in an aspect should be kept simple. In particular, the presentation rules should allow suppressing details. Complete parameter lists, visibility, raised exceptions and other details may never be specified in the architecture, especially when the design is not input to code generation or when an "agile" process is in place.

A thorough investigation of practical issues in the representation of aspects in the architecture should look at other topics related to architecture representation:

- *Documenting behavior*: what notations best capture dynamic behavior in the architectural design? What UML extensions would be appropriate in sequence diagrams, state machines and activity diagrams?
- *Platform independent aspects*: what provisions should be made in the notation to properly accommodate features and restrictions of the different AOP platforms and languages that exist?[2] Would an MDA-like approach (platform independent AO model with transformation rules to create a platform specific AspectJ model) be useful?
- *Aspect modules and crosscutting concerns*: in the architecture, when and how should we make the distinction between the crosscutting concerns that will be realized as aspects at the programming level and those that will not?
- *Tool support*: what are the benefits and restrictions of using a tool such as IBM Rational Rose and Omondo EclipseUML for creating design diagrams in comparison with a drawing tool such as Microsoft Visio? How does the purpose of the architecture documentation (reading reference vs. input to code generation) affect the right balance of the amount of detail added to diagrams?
- *Static AOP*: how should we represent static AOP constructs (e.g., member introductions and type hierarchy modifications [12]) in module views of the architecture using UML? Are there cases in which compile time declarations and exception softening are worth representing at the architectural level?

[2] This paper is biased towards an understanding of AOP based on the AspectJ language.

6. REFERENCES

[1] Bass, L., Clements, P. and Kazman, R. *Software Architecture in Practice, 2nd ed.* Addison-Wesley, 2003.

[2] Clements, P., et al., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2002

[3] *UML 2.0 Superstructure Specification*. OMG, 2004.

[4] Suzuki, J., and Yamamoto, Y. “Extending UML with Aspects: Aspect Support in the Design Phase.” Proceedings of AOP Workshop at ECOOP, 1999.

[5] Stein, D., Hanenberg, S., and Unland, R. “A UML-based Aspect-Oriented Design Notation For AspectJ”. Proceedings of the 1st International Conference on AOSD, 2002

[6] Jacobson, I., and Ng, P. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, 2005.

[7] Kandé, M., Kienzle, J., and Strohmeier, A. “From AOP to UML—A Bottom-Up Approach.” Workshop on aspect-oriented modeling with UML at the AOSD conference, 2002.

[8] Pawlak, R. et al. “A UML Notation for Aspect-Oriented Software Design.” Workshop on aspect-oriented modeling with UML at the AOSD conference, 2002.

[9] Zakaria, A., Hosny, H, and Zeid, A. “A UML Extension for Modeling Aspect-Oriented Systems.” Workshop on aspect-oriented modeling with UML at the AOSD conference, 2002.

[10] Aldawud, O., Elrad, T., Bader, A. “Uml Profile for Aspect-Oriented Software Development.” Proceedings of Third International Workshop on Aspect-Oriented Modeling, 2003.

[11] Lesiecki, N. “Applying AspectJ to J2EE Application Development.” Practitioner report, AOSD conference, 2005.

[12] Laddad, R. *AspectJ in Action*. Manning, 2003.