Extending UML for Modeling Reflective Software Components

Junichi Suzuki and Yoshikazu Yamamoto

Department of Computer Science,
Graduate School of Science and Technology,
Keio University
Yokohama City, 223-8522, Japan.
+81-45-563-3925 (Phone and FAX)
{suzuki, yama}@yy.cs.keio.ac.jp
http://www.yy.cs.keio.ac.jp/~suzuki/project/uxf/

Abstract. This paper describes our extension of the UML metamodel for specifying reflective software components. Reflection is a design principle that allows a system to have a representation of itself in the manner that makes it easy to adapt the system to a changing environment. It has matured to the point where it is used to address real-world problems in various areas. We describe how to document reflective components in the framework of UML. Our work allows for recognizing and understanding reflective components in the upper levels of abstraction at an earlier stage of the development process. It leverages the documentation, learning, visual modeling, reuse and roundtrip development of metalevel designs. We also demonstrate the seamless model exchange between different development tools and model continuity across development phases with application-neutral interchange formats.

1 Introduction

The Unified Modeling Language (UML) [1] has been widely accepted as a standard modeling language in the software engineering community. It defines semantics and their notations of model elements required for documenting object oriented software. It is a single and universal language that can be used with any design methodology. UML provides nine diagrams with fine level of abstraction to specify object models for a given problem. Complex systems can be modeled through a small set of nearly independent diagrams. UML has been used for representing a variety of software models such as real-time systems, hypermedia, business processes, engineering design and multiagent models.

This paper describes an extension of the UML metamodel for supporting reflective software components, which allows software to be highly configurable and extensible, and how to specify them with UML. Supporting reflective components in the design level means that we can recognize them in the upper levels of abstraction at an earlier stage of the development process. We also address a description language based on XML (eXtensible Markup Language) for describing reflective components as textual representations. It increases the model

continuity across development phases in more precise manner and provides for the interchangeability of reflective model information between different development tools.

The remainder of this paper is organized as follows. Section 2 overviews reflective software and its general constructs; it also describes the benefits of capturing reflective components in the design phase and expressing them with XML. Section 3 describes our extension to the UML metamodel and shows some examples. Section 4 presents some applications using UXF (UML eXchange Format), our XML-based model description language, and XMI (XML Metamodel Interchange) format. We conclude with our current project status and future work in Sections 5 and 6.

2 Reflective Components in the Design Phase

This section overviews Aspect-Oriented Programming. It then considers some aspects of reflective software and describes our motivation to capture its components in UML. We also describe an interchange format for reflective components.

2.1 Aspect-Oriented Programming: Separation of Concerns

Today's software is becoming more and more complex, and has to deal with an greater variety of computing concerns simultaneously, e.g. concurrency, object interaction, persistence, distribution, fault tolerance and realtime constraints. The notion of *separation of concerns* is proposed for managing software complexity well and improving its quality by separating different concerns and introducing clear and minimal dependencies between them at both the conceptual and implementation levels [2].

Aspect-oriented Programming (AOP) is a paradigm for facilitating separation of concerns, which has been proposed by a research group of the Xerox PARC [3]. AOP introduces a new unit of software modularity, called an *aspect*, which represents a computing concern described above. Each aspect provides a better handle for managing *cross-cutting* problems. Cross-cutting is a problem found in many object-oriented software systems; some features of a system, i.e. aspects in the sense of AOP, tend to affect or require the collaboration of groups of objects. They are naturally spread within a whole system and cross-cut the primary decomposition of objects. Typically, non-functional concerns make it difficult to understand and evolve the system.

Aspects are handled to fulfill certain application requirements (e.g. persistence, distribution, real-time and fault tolerance) or manage and optimize underlying computational algorithms (e.g. concurrency and object interaction). Isolating aspects allows them to be:

- abstracted to a higher level,
- easier to understand because an aspect's code is not cluttered with the code of other aspects, and

– coupled loosely with each other; thereby the flexibility and reusability of an aspect is increased.

The process of combining an aspect with other aspects or other portions of a system is called *aspect weaving*. A tool for weaving aspects is called *aspect weaver*. An aspect is expressed by encoding the aspect support as a conventional library, designing a separate language for the aspect, or designing a language extension for the aspect [4]. Aspect weaving is performed by source code transformation, component composition or reflection [4]. For example, AspectJ [5], an aspect weaver extending Java, represents an aspect by designing a language extension and weaves aspects by source code transformation. AOP/ST [6], an aspect weaver extending Smalltalk, represents an aspect by designing a separate language for the aspect and weaves aspects by reflection.

2.2 Reflection

We are using the reflection mechanism to separate aspects and keep them loosely coupled. Alternative approaches are adaptive programming [7] and component composition mechanisms [8, 9].

Reflection is a design principle that allows a system to have an explicit representation of itself in a manner that makes it easy to adapt the system to a changing environment [10]. It was originally introduced by 3-Lisp [11]. After that, it has been studied within various programming languages such as CLOS [12], Smalltalk [13], C/C++ [14] and Java [15], in order to extend the language syntax and semantics by providing language constructs as self-representations [12]. Recently, it has been applied to more generic system designs such as databases, concurrent/parallel computing, operating systems, virtual machines, distributed computing, security and agent-based intelligent systems. Reflection has matured to the point where it is used to address real-world problems. In fact, it is identified as a pattern of software architecture (POSA) [16].

In object-oriented systems, the base unit of computation is object (or *baseobject*). Through the interaction among objects, a system computes a certain task. Reflection introduces the notion of object/metaobject separation. A *metaobject* (or *metalevel object*) is an object that contains information about the internal structure and/or behavior of one or more baseobject. In other words, metaobjects can track and control certain aspects (i.e. structure and/or behavior) of baseobjects. A set of metaobjects is called a *metalevel*, and a set of baseobjects is called a *baselevel*.

Reflection is the ability of a program to manipulate as data something that represents the state of the program [17], and to adjust to changing requirements. The goal of reflection is to allow a baseobject to reflect on its own execution state and eventually alter it to change its meaning. In contrast to reflection, reification [18] is the process of making something accessible that is normally unavailable in the baselevel or is hidden from the programmer. For the execution of a baseobject to be supervised, it must first be reified into the corresponding

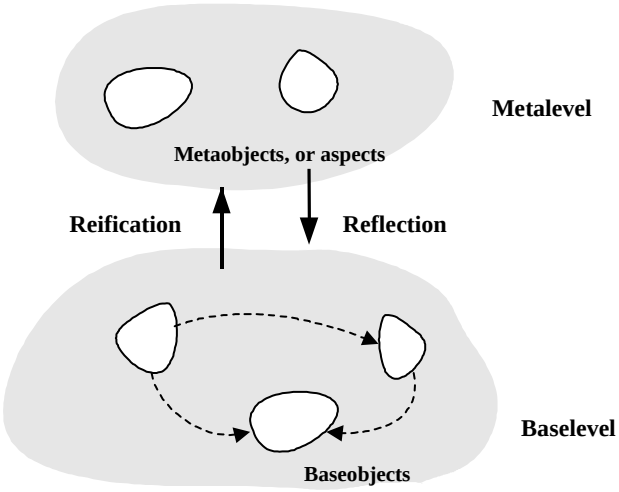

Fig. 1. Typical reflective architecture

metalevel. A set of interfaces through which a baseobject interacts with its metalevel is called *Metaobject Protocols* or *MOPs* [12]. The relationship among the constructs described above is illustrated in Figure 1.

In general, a metaobject protocol establishes the following interactions [19]: (1) attachment of baselevel and metalevel objects, that can be static or dynamic, and in one-to-one or many metaobjects to one baseobject basis; (2) reification of structural and/or behavioral features within a baseobject; (3) execution, which consists of metalevel computation that interferes with the baselevel behavior transparently through the interception and reification mechanisms; (4) modification, which is the capability of the metaobjects of changing behavior and structure of baseobjects.

There are various approaches to achieve reflection, and they can be classified into: compile-time and runtime reflection [14], structural and behavioral reflection [13], and introspection and intercession [20].

In our work, a metaobject is considered as an entity representing an aspect in the sense of AOP. The separation of concerns is performed by separating metaobjects from baseobjects and keeping metaobjects isolated. Aspects and metaobjects are called reflective components.

2.3 Benefits of Capturing Reflective Components in the Design Phase

Reflective components can be identified at the design and implementation phases, though the cross-cutting problem tends to occur at the implementation/coding phase [4]. When a reflective component is identified, or emergent, at the implementation phase, developers often add it to a system and change existing components manually (see Figure 2). Reflective components are maintained only at

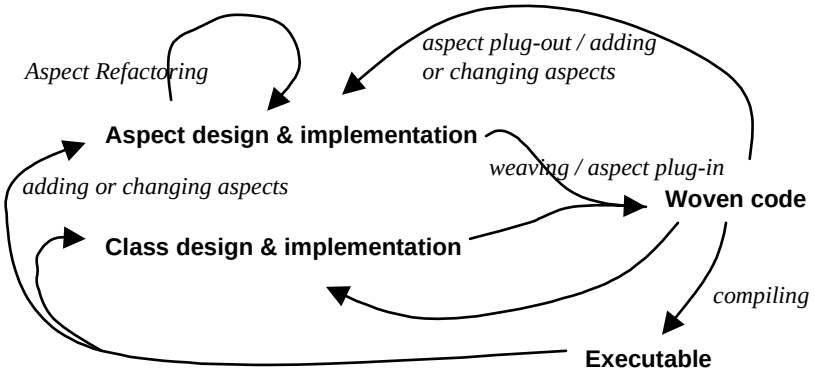

Fig. 2. Typical process of aspect-oriented software development

the source code level. Few methods have been proposed for expressing them at the design level. Supporting them at the design phase streamlines the process of reflective system development (Figure 2) and increases the productivity of metalevel designers by facilitating:

- Documentation and Learning
  Supporting reflective components as design constructs allows developers to recognize them in the upper level of abstraction at an earlier stage of the development process. Metalevel designers can easily document and understand reflective system design.
- Visual modeling
  Metalevel designers can understand reflective components in more intuitive way by visualizing them using CASE tools or model viewers.
- Reuse of metalevel design
  The above characteristics leverage the reuse of the metalevel design. A promising example is pattern catalogs that collect well-known and feasible design of reflective components.
- Roundtrip development
  The incremental and roundtrip development of reflective software is possible with the metalevel source code to design model translation, model to metalevel source code translation and metalevel refactoring (see also Figure 2).

2.4 Benefits of Describing and Interchanging Metalevel Design Models with XML

In the software design phase, model interchange is a very important capability, because there are few application-neutral model exchange formats between development tools. To solve this problem, we developed the UML eXchange Format (UXF) [21], which is similar in some respects to the XML Metamodel

Interchange (XMI) format [1], standardized by the Object Management Group. Both UXF and XMI are based on XML and allow model semantics to be described explicitly and transferred precisely. Such application-neutral interchange format facilitates:

- Interchangeability and reuse of metalevel descriptions:
  Software models change dynamically in the analysis, design, implementation and maintenance phases. Software tools used in each phase usually employ their own proprietary formats to describe model information. For example, current aspect weavers use their own language to describe aspects. An application-neutral format allows aspect information to be interchangeable and reusable between a wide range of different development tools with different strengths, throughout the lifecycle of software development (see also Figure 2). This seamless tool interoperability increases our productivity for designing the metalevel.
- Intercommunication between metalevel designers:
  An application-neutral metalevel description format serves as a communication vehicle for designers. They can communicate their modeling insights, understandings and intentions on a metalevel design with each other. This capability simplifies the circulation of aspect models between aspect designers.

We use UXF basically to describe metalevel design information, because it preceded XMI at the time when our project began. However, our project is slowly migrating to use XMI by providing a UXF-XMI converter and extending XMI. Note that due to space limitations, the basics and benefits of using XML as an interchange format are not covered here. Please see [?] for a more in-depth discussion.

3 Modeling Reflective Components with UML

This section describes our extensions to the UML metamodel for supporting reflective components and how to specify them with UML.

3.1 Aspects

The semantics of an aspect is defined as a UML metamodel element derived from `Classifier`, which describes behavioral and structural features [1]. As shown in the left of Figure 3, `Classifier` is a parent element of `Aspect`, `Class`, `Interface`, `Node` and `Component`.

An aspect can have a set of attributes, operations and relationships because a classifier has them. An operation of an aspect is considered as the aspect's weave declaration. A weave is a program that centralizes the code affecting (crosscutting) diverse portions in a system. An attribute of an aspect is used by one or more weaves. Relationships of an aspect include generalization, association and

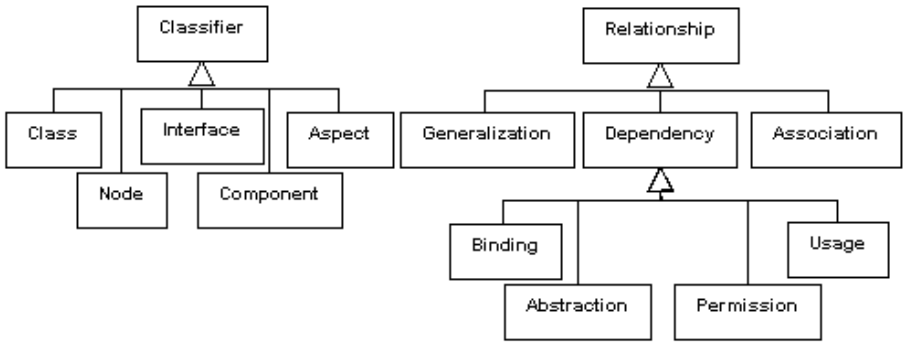

Fig. 3. `Aspect` as a metamodel element derived from `Classifier` (left), and some kinds of the `Relationship` metamodel element(right)

dependency (see Figure 3). If an aspect language and weaver supports multiple kinds of weaves, e.g. introduction and advice weaves in AspectJ [5], they are specified as stereotypes corresponding to their kinds.

The notation of an aspect is a class rectangle with stereotype ≪aspect≫ as shown in Figure 4. In this figure, the `Singleton` aspect represents the Singleton design pattern. Because the aspect encapsulates the instantiation and instance management policies, it is possible for a class to vary the policies. The operation list compartment of the rectangle means the list of weave declarations. Each weave is displayed as an operation with the stereotype ≪weave≫. A signature of a weave declaration shows its designator, specifying which model elements (e.g. classes, methods and variables) are affected by the weave.

The `Singleton` aspect in Figure 4 is defined based on the AspectJ language. It has two introduction weave declarations specified by the stereotype ≪introduction weave≫. `Singleton` introduces a static method `GetInstance()` and a private constructor in `SingletonClass`. A static attribute `Instance`, which is stereotyped with ≪introduction≫ is also introduced in `SingletonClass`.

3.2 Aspect-Class Relationship

UML defines three primary relationships derived from the `Relationship` metamodel element: `Association`, `Generalization` and `Dependency` (Figure 3). The relationship between an aspect and the classes affected by the aspect is a kind of dependency, because the behavior of a class is constrained by an aspect. The dependency relationship states that the implementation or functioning of one or more elements requires the presence of one or more other elements [1]. The derived metamodel elements of `Dependency` are `Abstraction`, `Binding`, `Permission` and `Usage` (Figure 3). The aspect-class relationship is classified as a kind of the abstraction dependency. The abstraction dependency relates two elements that are the same concept at different levels of abstraction or from different viewpoints [1]. UML defines three stereotypes for the abstraction de-

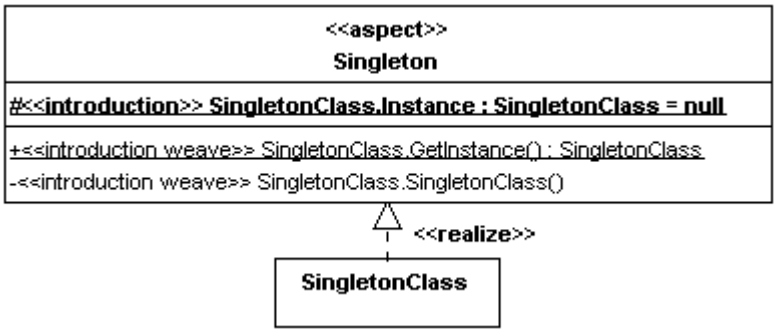

Fig. 4. A simple aspect example. The `Singleton` aspect reifies the implementation policies of the Singleton design pattern.

pendency: *derivation*, *realization*, *refinement* and *trace*. The aspect-class relationship is best-suited to the abstraction dependency with the stereotype realization, ≪realize≫. A realization is a relationship between a specification model element and a model element that implements it. The implementation model element is required to support the declaration of an specification model element.

UML defines the notation for an abstraction dependency with the ≪realize≫ stereotype as a dashed generalization arrow. Figure 4 shows a single aspect-class relationship between `Singleton` and `SingletonClass`.

3.3 Woven Class

Aspect and class code are combined using an aspect weaver, and then a woven class is generated (Figure 2). The woven class structure depends on the aspect weaver and programming language used. For example, AspectJ replaces an original class with the generated woven class. AOP/ST generates a woven class derived from the original class [6]. There are other alternative composition strategies, e.g. composition using the Mediator or Decorator design patterns [22].

We introduced the stereotype ≪woven class≫ into the `Class` element in order to represent a woven class. Figure 5 shows two different woven class structures using different aspect weavers, AspectJ and AOP/ST. In general, application developers may not use this stereotype because they do not have to recognize how a woven class is structured. However, since aspect or metalevel designers often need to know the implementation details of weavers, they can use it especially when they remove an aspect and modify the aspect and/or classes to debug them (see also Figure 2).

3.4 Example: Reifying the Observer design pattern in the metalevel

Figure 6 shows an example of an aspect-oriented variant of the Observer design pattern [22] based on the AspectJ language. Reifying design pattern con-

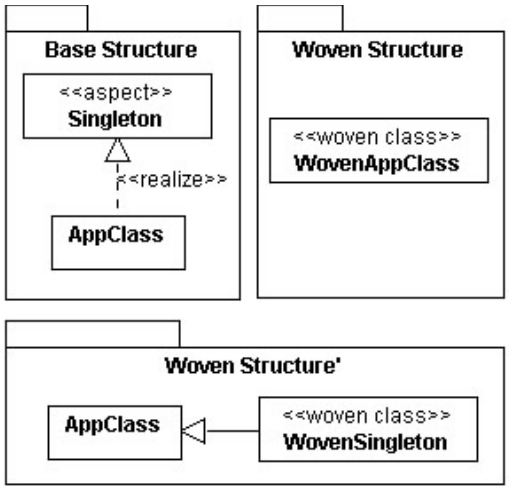

Fig. 5. An aspect-class structure and two different woven class structures using AspectJ (upper) and AOP/ST (bottom).

structs to the metalevel allows for centralizing a variety of implementation policies in a pattern, as proposed in several places [15, 23, 24]. In Figure 6, the aspect `SubjectObserverProtocol` reifies the behavior of the pattern to define the behavior between the class `Subject` and `Observer`. `SubjectObserverProtocol` can be implemented independently of `Subject` and `Observer` because it localizes a policy of the protocol implementation involving several objects in a single aspect rather than spreading multiple code fragments throughout them.

`SubjectObserverProtocol` defines seven introduction weaves and two attributes. The following definitions:

```
#<<introduction>> Observer.subject: Subject=null
+<<introduction weave>> Subject.notify(arg: Object):void
```

are mapped to the AspectJ aspect definition below:

```
aspect SubjectObserverProtocol{
  introduction Observer{
    protected Subject subject = null;
    ...
  }
  introduction Subject{
    public void notify(Object arg){
      ...
    }
  ...
}
```

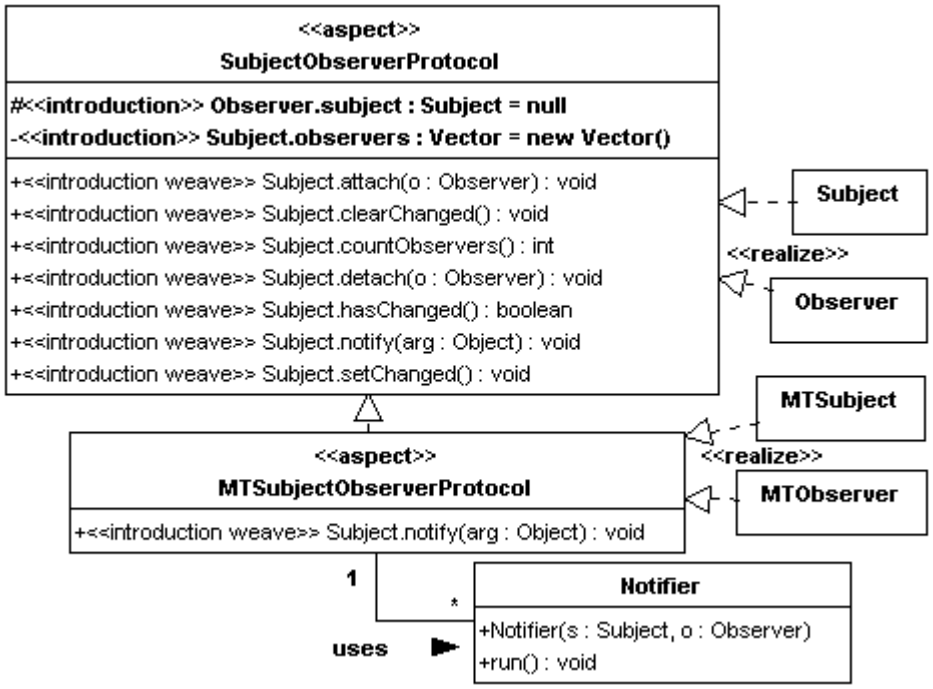

Fig. 6. Aspects reifying the Observer design pattern.

`SubjectObserverProtocol` has a sub-aspect, `MTSubjectObserverProtocol`, which is a thread-safe aspect. The implementation policy of a design pattern can be modified by extending an aspect. `MTSubjectObserverProtocol` is used when a subject and observers are executed on different threads. It is designed to avoid a potential deadlock caused when a subject issues a change notification in a thread while an observer is trying to check the observable's instance variable [25]. `MTSubjectObserverProtocol` uses an instance of `Notifier` to issue an event in a new different thread. Notifier is created for a single event notification to an observer. A new thread spawned to execute `run()` of `Notifier` does not possess the synchronization lock on a `MTSubjectObserverProtocol`. Note that `MTObserver` and `MTSubject` are not subclasses of `Observer` and `Subject`. They can support any other arbitrary implementation and/or interfaces.

3.5 Example: An Aspect-Oriented Web Server

The next example shows an aspect layer (i.e. metalevel) in our adaptive web server named OpenWebServer [26]. OpenWebServer contains a metalevel that supports a wide range of aspects of web servers to allow itself to continuously evolve beyond the static and monolithic servers of today. It has a collection of aspects including `Concurrency`, `Cache`, `Protocol`, `RequestHandler`,

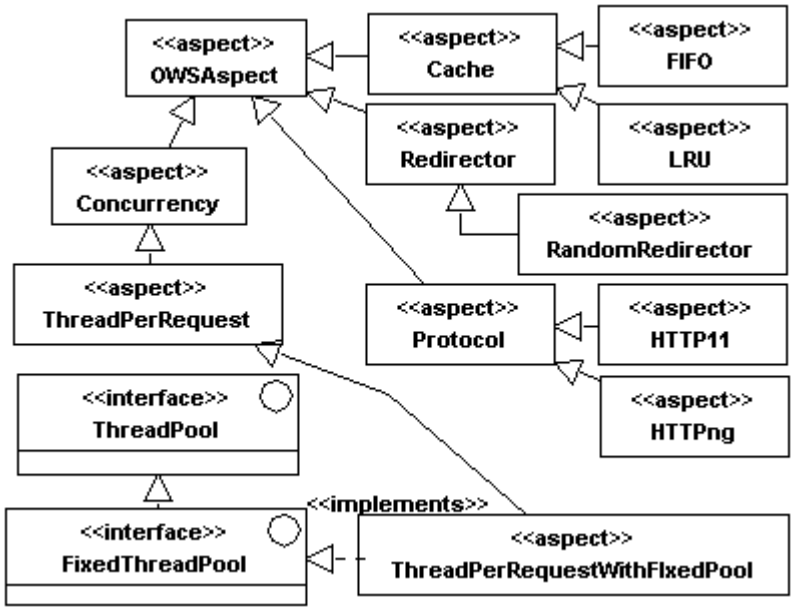

Fig. 7. Aspects in OpenWebServer

`ContentFinder`, `Logger`, `Redirector` (see Figure 7). Each aspect has one or more implementation policies. OpenWebServer can change its behavior at runtime corresponding to a given situation and/or requirement. The system adaptation is achieved with the reflection mechanism. In our project, project members are using the semantics and their notations described above to specify aspects and keep them loosely coupled in both conceptual and implementation levels.

3.6 Metalevel, Baselevel and Metaclasses

All of metalevel, baselevel, metaclasses and baseclasses can be represented using the predefined UML model constructs.

The metalevel and baselevel are expressed with the `Package` element. A metalevel is described with a package with the stereotype ≪metamodel≫. A class stereotyped with ≪metaclass≫ represents a metaclass. The relationship between a class and its metaclass is described with the `InstanceOf` relationship.

Figure 8 shows an example of a reflective implementation of the proxy design pattern, described in [27]. This model defines a *proxy* object [22] that hides the reference to an object and plays the role of its placeholder to control access it. This pattern is generic enough to provide various kinds of proxies including virtual proxy, cache proxy, remote proxy, protection proxy, synchronization proxy, counting proxy and firewall proxy [28]. Figure 8 defines local and remote proxies (`ProxyForX` and `ProxyForY`) for original (or target) objects (`X` and `Y`). Both proxies accept a method invocation and redispatch it to corresponding

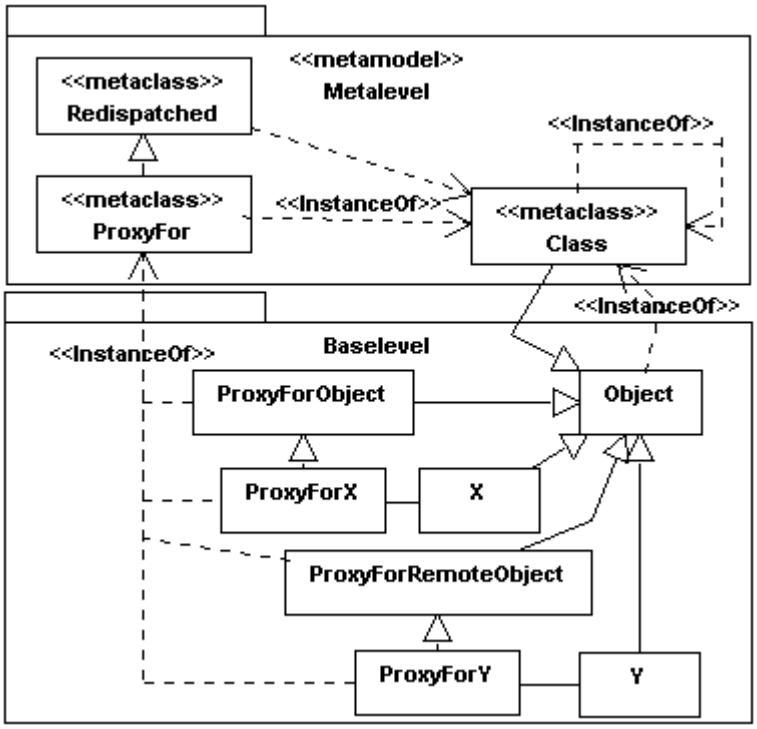

Fig. 8. A reflective implementation of the proxy design pattern

target objects. A remote proxy creates the illusion that the remote object is a local one. `Redispatched` provides this redispatching capability by placing a redispatch stub in each method table entry with its `redispatch()`. Therefore, the metaclass for proxies, `ProxyFor`, is a subclass of `Redispatched`. The method `redispatch()` must be overridden in base proxy classes, `ProxyForObject` and `ProxyForRemoteObject`, for implementing redispatch stubs according to the kinds of proxies. `Object` is a base class for all classes and `Class` is a base class for all metaclasses. Note that, as in the example in Figure 6, `X` is not a parent of `ProxyForX`; instead `ProxyForX` merely has to support the interface of `X` [27].

The notation of the `InstanceOf` relationship is a dashed arrow with its tail on a class and its head on a metaclass. The arrow has the keyword ≪instanceOf≫. We introduced alternative notations for the relationship between a class and its metaclass, ≪reflect≫ and ≪reify≫. ≪reflect≫ is a directed association with its tail on a metaclass and its head on a class. ≪reify≫ is a directed association with its tail on a class and its head on a metaclass. These associations are useful when a programming language does not support metaclasses directly as language constructs.

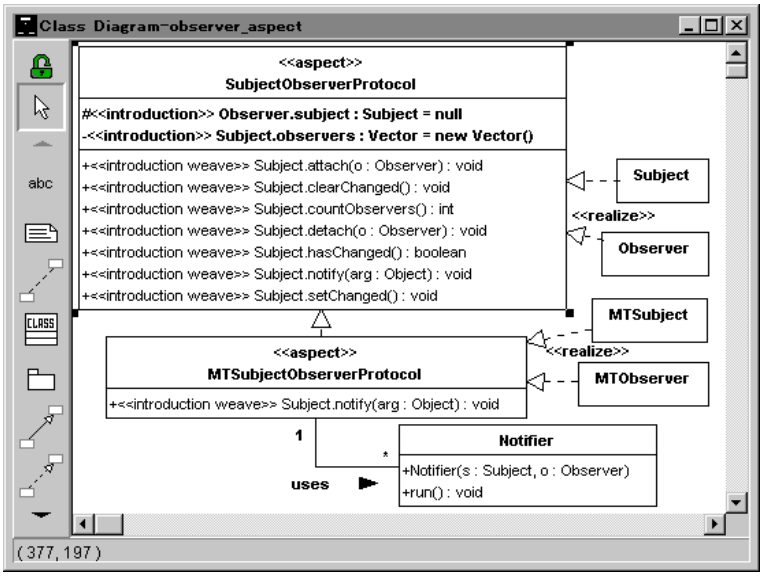

Fig. 9. A sample screen display of MagicDraw showing an aspect model information generated from a UXF description (The graphical positions of the icons are changed manually).

4 Describing and Interchanging Reflective Components with UXF and XMI

As described in Section 2.4, we are using UXF basically to describe metalevel model information. We developed a UXF DTD (Document Type Definition) for metalevel description and then merged it with existing UXF DTDs.

We have developed a translator that converts an aspect source code of AspectJ into a UXF description and vice versa. For translating an aspect code to a UXF description, the translator parses AspectJ code using the Doclet toolkit included in Java Development Kit. For translating a UXF description to an aspect code, the translator parses UXF code with an XML parser and generates AspectJ code. The aspect model interchange can be performed at the aspect design/coding, class design/coding and woven class verification phases (see Figure 2). It helps the forward/reverse engineering in roundtrip aspect development.

We have also developed a tool to convert a UXF description into native formats of Rational Rose [1] and MagicDraw [2], which are popular commercial CASE tools. Figure 9 is a screen display of MagicDraw showing an aspect model generated from the UXF description, which is in turn translated from an AspectJ code (see also Section 3.4 and Figure 6). This example shows the tool interoperability and aspect model interchangeability between an aspect weaver and a CASE

[1] http://www.rational.com/rose/
[2] http://www.nomagic.com/

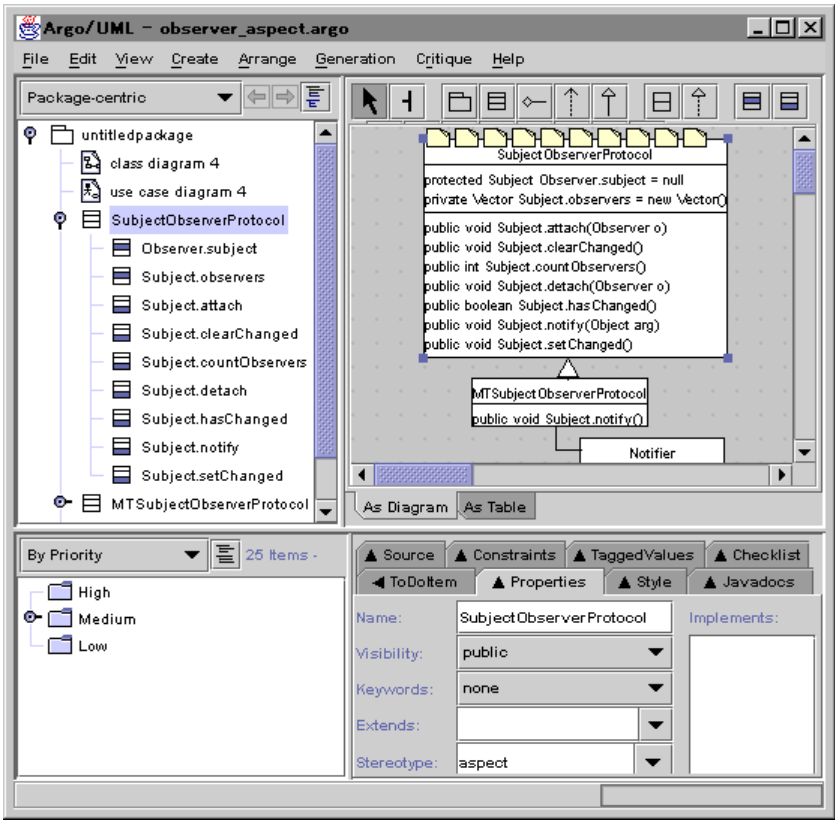

Fig. 10. A sample screen display of Argo/UML showing an aspect model information generated from our UXF-XMI converter (The graphical positions of the icons are changed manually).

tool. In addition, we are developing a UXF-XMI converter for XMI-enabled tools to use UXF descriptions. Figure 10 is a screen display of Argo/UML [3], which is an XMI-enabled CASE tool, showing an XMI formatted model description converted from UXF. This example shows the tool interoperability and model interchangeability between different CASE tools.

5 Current Project Status and Future Work

Our project is using AspectJ as an aspect language and Java as a programming language. We are now evaluating the interoperability of aspect model information between weavers.

[3] http://www.ics.uci.edu/pub/arch/uml/

For the purpose of interchanging the semantics of aspect models, we are investigating an alternative approach that deals with aspect constructs in a separate metamodels based on the Meta Object Facility (MOF) [29].

As for our UXF-XMI converter, it supports only the conversion from UXF's class diagram descriptions into XMI's class diagram descriptions. We plan to support the conversion for all the model constructs.

For the smooth transition in the lifecycle of aspect-oriented development, we started to develop an aspect refactoring tool, which supports common refactoring operations automatically, instead of manually by programmers. Typical transformation in the aspect refactoring includes dividing an aspect into two aspects (e.g. new sub-aspect, super-aspect, abstract aspect) and merging one or more aspects into an aspect.

6 Conclusion

This paper describes how to represent reflective architectures in the framework of UML. Our work allows us for recognizing and understanding reflective components in the upper levels of abstraction at an earlier stage of the development process. It leverages the documentation, learning, visual modeling, reuse and roundtrip development of metalevel designs. We also demonstrate the seamless model exchange between different development tools and model continuity across development phases with application-neutral interchange formats.

7 Acknowledgements

We sincerely thank Eduardo B. Fernandez and and UML'99 referees for improving this paper with their careful reading and invaluable comments.

References

1. Object Management Group. *Unified Modeling Language Specification version 1.3.* OMG document number: ad/99-06-08, 1999.
2. W. L. Hursch and C. V. Lopes. Separation of Concerns. Technical report, NU-CCS-95-03, Northeastern University, 1995.
3. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP'97.* Springer LNCS 1241, 1997.
4. K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering based on Automated Configuration and Fragment-based Component Models.* Ph.D. Thesis, Technische Universitat Ilmenau, Germany, 1998.
5. Xerox PARC AOP research group. The AspectJ Primer. available at www.parc.xerox.com/spl/projects/aop/aspectj/primer.
6. K. Bollert. Implementing an Aspect Weaver in Smalltalk. In *Proceedings of STJA'98*, 1998, available at www.germany.net/teilnehmer/101,199268/.
7. K. J. Lieberherr. *The Art of Growing Adaptive Object-Oriented Software.* PWS Publishing Company, 1995.

8. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In R. Guerraoui O. Nierstrasz and M. Riveill, editors, *Object-based Distributed Processing*. Springer LNCS, 1993.
9. Y. Ichisugi and Y. Roudier. The Extensible Java Preprocessor Kit and a Tiny Data Parallel Java. In *Proceedings of ISCOPE'97*. Springer LNCS 1343, 1997.
10. S. Sonntag, H. Haertig, O. Kowalski, W. Kuehnhauser, and W. Lux. Adaptability using Reflection. In *Proceedings of 27th. Annual Hawaii Int. Conf. on Sys. Sci.*, pages 383–392. Springer LNCS 1616, 1994.
11. B. C. Smith. Reflection and Semantics in Lisp. In *Proceedings of ACM POPL '84*, pages 23–35, 1984.
12. G. Kiczales, J. Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.
13. J. McAffer. Engineering the Meta Level. In *Proceedings of Reflection '96*, 1996.
14. S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA'95*, 1995.
15. M. Tatsubori and S. Chiba. Programming Support of Design Patterns with Compile-time Reflection. In *Proceedings of Reflective Programming in C++ and Java Workshop at OOPSLA'98*, 1998.
16. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. WILEY, 1996.
17. P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA '87*, pages 147–155, 1987.
18. D. P. Friedman and M. Wand. Reification: Reflection without Metaphysics. In *Symposium on LISP and Functional Programming*, 1984.
19. M.L.B Lisboa. A New Trend on the Development of Fault-Tolerant Applications: Software Meta-Level Architectures. In *Proceedings of Internetional Workshop on Dependable Computing and its Applications (IFIP'98)*, 1998.
20. D. Bobrow, R. Gabriel, and J. White. CLOS in Context -The Shape of the Design Space. In A. Paepcke, editor, *Object-Oriented Programming- The CLOS Perspective*, page Chapter 2. MIT Press, 1993.
21. J. Suzuki and Y. Yamamoto. Making UML Models Interoperable with UXF. In J. Bezivin and P-A. Muller, editors, *≪UML≫'98: Beyond the Notation*. Springer LNCS 1618, 1999.
22. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
23. C. Maros, M. Campo, and A. Pirotte. Reifying Design Patterns as Metalevel Constructs. In *Journal of the Argentine Society for Informatics and Operations Research*. August 1999.
24. L. L. Ferreira and C. M. F. Rubira. The Reflective State Pattern. In *Proceedings of PLoP'98*, 1998.
25. D. Lea. *Concurrent Programming in Java: Design Principle and Patterns*. Addison-Wesley, 1997.
26. J. Suzuki and Y. Yamamoto. OpenWebServer: an Adaptive Web Server Using Software Patterns. In *IEEE Communications Magazine, Vol.37, No.4*, April 1999.
27. I. R. Forman and S. H. Danforth. *Putting Metaclasses to Work*. Addison-Wesley, 1998.
28. H. Rohnert. The Proxy Design Pattern Revisited. In J. Vlissides J. Coplien and N. Kerth, editors, *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.
29. Object Management Group. *Meta Object Facility*. OMG document number: ad/97-08-14, November 1997.