DESIGN PATTERN SPECIFICATIONS IN ASPECT-ORIENTED EXECUTABLE UML

Shinya Kosuge, Akira Teruya, Eiichiro Iwata
Masahito Sugai, Noriko Matsumoto, Norihiko Yoshida
Department of Information and Computer Sciences
Saitama University
255 Shimo-Ohkubo, saitama 338-8570, JAPAN

ABSTRACT

A program can be described in a form of high reusability by using design patterns. However, since codes about a pattern straddle multiple classes, it is difficult to apply the pattern to the program which is created without using design patterns. Although this problem can be solved by separating the codes about a pattern using Aspect-Oriented Programming (AOP), such a research was only done using AspectJ language. So, its description depends on a specific language. Hence, this paper proposes describing it using Executable UML whose abstract degree is higher than other languages. As a result, the drawback that the design pattern by AOP depends on the specific language can be solved, and the design pattern can be treated as parts regardless of a platform.

KEYWORDS

Design patterns, Executable UML, Aspect-Oriented Programming

1. INTRODUCTION

A design pattern is to promote reuse of a structure and a function which often appear in several applications. Its advantages are not only efficiency improvement in software development by reuse, but also communication aids between programmers. However, it has mainly two types of drawbacks.

First, a design pattern is described in a natural language and sample codes written in a specific programming language. Therefore, it is to describe strict behaviors, and to understand how to implement. Sample codes help us to understand its implementation, however, it depends on the specific programming language. In order to solve this dependence, Mukasa et al. [Mukasa, 2007] proposed the way to describe the strict behavior by using Executable UML (xUML) and Model Driven Architecture. In the proposal, the strict behavior is described without using the sample codes in a specific language.

The second drawback is that codes in a design pattern are concerned with multiple classes. Therefore, many classes must be tailored when applying a design pattern to a program which is created without the design pattern. This makes difficult to apply it automatically. Aspect-oriented languages are effective to solve this drawback. Hannemann et al. [Hannemann, 2002] proposed an attempt to use AspectJ based on Java, however, this proposal still depends on a specific programming language. Consequently, this paper proposes a description framework, which is independent of any specific programming language such as AspectJ, using xUML whose abstraction is higher than a programming language.

This paper is organized as follows. Section 2 summarizes Aspect-Oriented Programming (AOP) and AspectJ. Section 3 summarizes xUML. Section 4 explains the Aspect-Oriented xUML (AOxUML) which makes aspects applicable to xUML. Section 5 overviews related research: design patterns in AOP. Section 6 describes differences of aspects between AspectJ and xUML, and its solutions. Section 7 describes a categorization of the design patterns from the point of AOP, and shows some examples of design patterns in xUML and in AOxUML. Section 8 describes considerations on design patterns described in this research. Section 9 describes the conclusion and the future subject.

2. ASPECT-ORIENTED PROGRAMMING

Aspect-Oriented Programming (AOP) [Chiba, 2005; Nagase, 2004] is a new modularization technology that improves the drawbacks of Object-Oriented Programming. AOP can modularize crosscutting concerns that are code-straddling across multiple classes. Examples of crosscutting concerns are logging and lock/unlock. In Object-Oriented Programming, The codes relevant to these processings need to be described in multiple classes, and it is necessary to change multiple classes when its specification is changed. This severely degrades reusability. In AOP, such crosscutting concerns are separated from the classes using aspects. Aspects are integrated into classes by "weaving".

There are three elements in AOP: an advice, a pointcut, and a joinpoint. An advice is a chunk of a code, and a joinpoint is a position in a program where the aspect is woven. The pointcut is a set of the joinpoints which fulfills the specified conditions. An aspect is composed of advices and pointcuts.

Figure 1 shows an example of aspect description in AspectJ, which is an AOP extension to Java. This Logger aspect specifies an execution position of methodA in ClassA as the joinpoint by the pointcut "execution". The aspect is with the "before" descriptor, therefore the advice is executed just before the joinpoint.

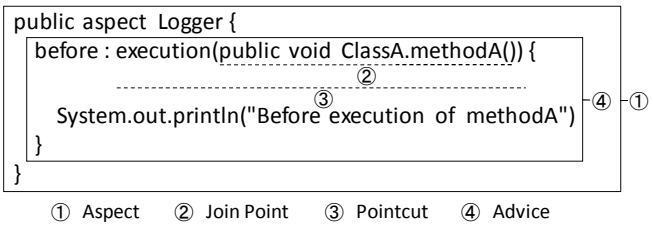

Figure 1. The example of description of the aspect in AspectJ

When this aspect is woven into ClassA, ClassA is transformed to ClassA' which executes the code described in the advice just before executing methodA. Figure 2 shows this conversion.

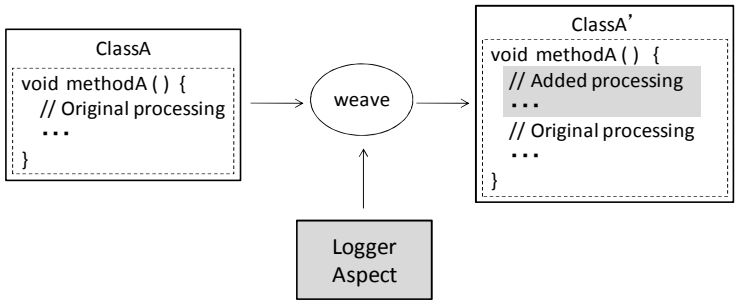

Figure 2. Weave of Logger aspect

AOP can be used not only for such structure conversion, but also for addition of a class, and change of inheritance relations.

3. EXECUTABLE UML

In the conventional UML, model behaviors cannot be described strictly nor verified, therefore, to test the UML models, they must be implemented in a programming language. Executable UML (xUML) [Mellor, 2002] is an extension to the conventional UML, and models in xUML can be executed and verified. xUML has a formal action language based on "UML Action Semantics [OMG, 2001]" of UML2.0 [OMG, 2000]. Strict description of behaviors is possible and the model can be verified using a model compiler. In this research, we use iUML [Kennedy Carter Ltd.] as a xUML tool.

4. ASPECT-ORIENTED EXECUTABLE UML

Teruya et al. [Teruya, 2008] proposed Aspect-Oriented Executable UML (AOxUML) which adds AOP technology to xUML. We use this system in our research, and this section explains this system.

4.1 Form of Aspect

This aspect processing system follows "JoinPoint Model (JPM)", therefore it handles joinpoints, pointcuts and advices. The effective joinpoints and advices for a xUML model are defined based on XMI that shows the structure of the xUML model.

Aspects are described in XML in this system. The weaver is implemented using an XML purser with some extension specified as XML tags and attributes. Figure 3 shows an example of the aspect description which conforms to the XML form.

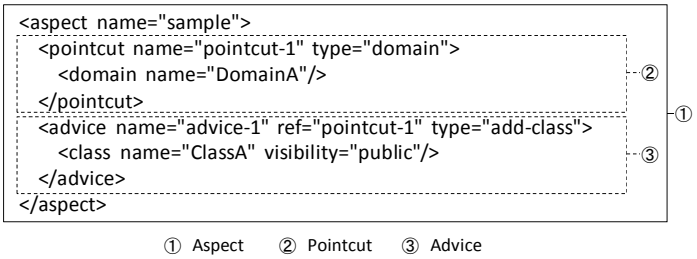

Figure 3. The description example of a single aspect

An aspect may contain more than one advices and pointcut. The pointcut consists of a pointcut name, a type of pointcut, and a definition body of pointcut. The type of pointcut describes a type of joinpoint specified as a position to weave in. The advice consists of an advice name, a type of advice, a pointcut name to apply advice, and a definition body of advice. The type of advice describes a type of processing performed in the joinpoint specified by the pointcut.

4.2 Weave of Aspect

Figure 4 shows the weaving procedure in xUML.

1) Transform an xUML model to a XML form.
2) Using the weaver, integrate XML of the model and XML of aspects..
3) Transform back the XML form of the model integrated with aspects to xUML.

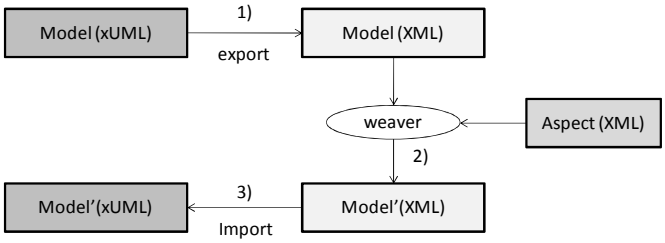

Figure 4. Outline of weaving aspect to xUML model

5. RELATED RESEARCH: DESIGN PATTERNS IN AOP

The codes for design patterns straddle the multiple classes, and it is difficult to modularize them in object-oriented programming. Hannemann et al. [Hannemann, 2002] solved this drawback by applying aspect-

oriented programming. Furthermore, a design pattern can be applied to programs automatically to some extent using a weaver.

However, Hannemann's work depends on a specific programming language, AspectJ. Our proposal uses xUML and is independent from any programming language

6. DIFFERENCE BETWEEN THE ASPECTS IN ASPECTJ AND XUML

There are some differences between the aspect in AspectJ and xUML, therefore it cannot describe by the same way at all. In this chapter, the differences and solutions are described.

- **Methods and fields in an aspect**
  Although AspectJ allows us to define a method and a field in an aspect, Aspect-Oriented xUML does not. This issue is solved by embedding a method and a field in a class which is added by the aspect. For example, the aspect for Observer design pattern in AspectJ defines "addObserver" as a method, while the aspect in this research embeds "addObserver" in the Subject class (See 7.3).
- **Abstract Pointcut**
  Although AspectJ allows us to define an abstract pointcut which does not specify any concrete joinpoint, Aspect-Oriented xUML does not. The abstract pointcut enables us to define only a process as an advice without specifying any concrete pointcut. This issue is solved by defining the process in the same manner as above.
- **The weave to a constructor**
  Although AspectJ allows us to define an initializing constructor using a pointcut, Aspect-Oriented xUML does not. Furthermore, the initialization processing of xUML cannot perform many things like the initializing constructor of Java. Therefore, it is difficult to describe patterns which are concerned with initialization of instances. This is one of our future works.

7. DESIGN PATTERNS IN ASPECT-ORIENTED XUML

In this research, we describe design patterns in xUML with AOP in order to solve the language-dependence problem.

Hannemann et al. [Hannemann, 2002] grouped design patterns into some categories as below. We specify some design patterns, each of which is out of each category respectively, in AOxUML, and examine features corresponding to each category. Due to the limited space, this paper only presents the Observer pattern as the most typical example.

7.1 The Categorization of the Design Patterns

Before presenting the design pattern in AOxUML, the categorization of design patterns is shown below.

- **Roles only used within a pattern aspect**
  The patterns classified in this category are Composite, Command, Mediator, Chain of Responsibility, and Observer. These patterns introduce roles which are only used within the pattern. In AOP, these roles are described using the aspect.
- **Aspects as object factories**
  The patterns classified into this category are Singleton, Prototype, Memento, Iterator, and Flyweight. These patterns have methods for creation and management of instances. These methods must be described in the aspect.
- **Language constructs**
  The patterns classified into this category are Adapter, Decorator, Strategy, Visitor, and Proxy. In these patterns, the structure of the pattern dissolves and is not obvious in the aspect.
- **Multiple inheritance**
  The patterns classified into this category are Abstract Factory, Factory Method, Template Method, Builder, and Bridge. In these patterns, the multiple inheritance features enables us to define aspects in a straightforward manner.

- **Scattered code modularization**
  The patterns classified into this category are State, and Interpreter. In these patterns, the codes straddling across the multiple classes are modularized by an aspect.
- **No benefit**
  The pattern classified into this category is Facade. This pattern gets no benefits from AOP.

7.2 Observer Pattern in xUML

Observer pattern enables an object to notify any state changes to another object automatically by calling its method.

Figure 5 shows the class chart of the Observer pattern described in xUML without an aspect. addObserver is an operation to register the Observer object. Since the attribute cannot hold an object of the class in xUML, we make a link to the object which should be hold. In this model, when setStatus is executed, notifyObservers is called from setStatus. notifyObservers calls update of Observer objects which are connected by the links. The action of update in Observer class is not defined in this class, but in the subclass of Observer class. We describe the operation to perform when setStatus is executed in this update.

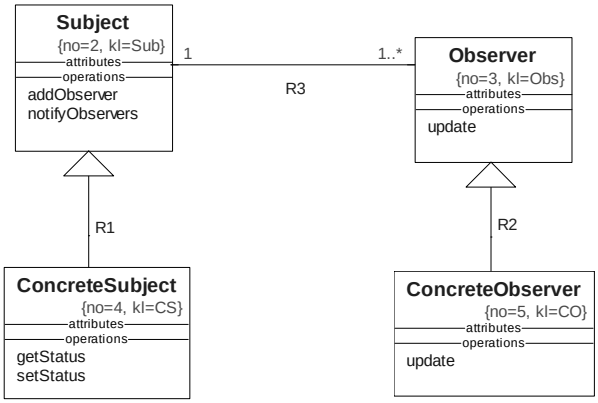

Figure 5. Class chart of Observer pattern that used iUML

7.3 Observer Pattern in AOxUML

In our design using AOxUML, all proccesings in the Observer pattern are modularized in an aspect. In addition, we describe the position of a call of notifyObservers in the aspect without describing it directly in an operation of a class.

The left of Figure 6 shows the class chart of the model before the aspects are applied. This model contains no class nor operation concerning the Observer pattern. Since the corresponding processing is not automatically done even if the value of the attribute is updated, it is necessary to describe the method call in every time. By weaving some aspects to this model, Observer pattern can be applied.

The right of Figure 6 shows the class chart of the model after applying Observer pattern. The portions enclosed by the dotted circles are the portions added by aspects common to Observer pattern, such as Subject class, Observer class, and operations of those classes. The other portions are the portions added by aspects which must be defined for every models, such as specification of the classes which inherit the Observer class and Subject class, positions of calling notifyObservers, and the processing to perform in update.

7.4 Other Patterns in AOxUML

We described some patterns of each category besides the observer pattern. As a result, we were able to confirm that “Roles only used within a pattern aspect”, “Language constructs”, and “Scattered code modularization” could be well described using an aspect also in xUML. It is useful to use an aspect in

"Aspects as object factories", but there are a few patterns that can be described in xUML. In "Multiple inheritance", the advantage using an aspect is lost in xUML.

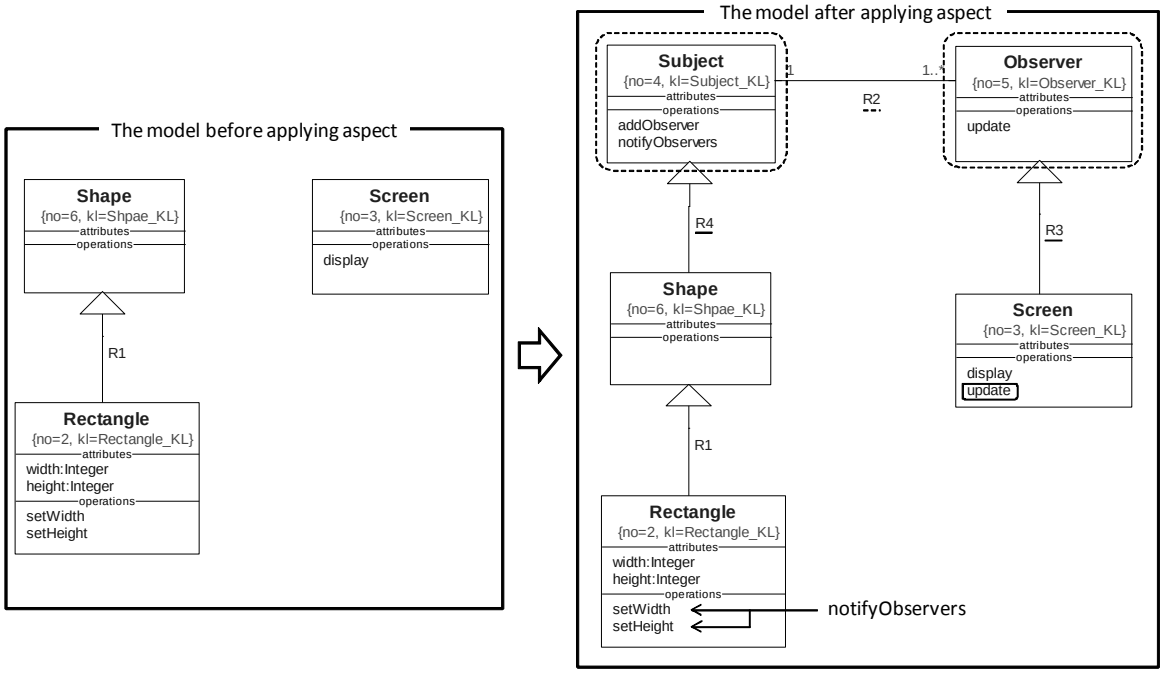

Figure 6. Example of applying Observer pattern

8. CONCLUSION

In this paper, we presented design patterns in xUML with AOP. As a result, design patterns are defined with AOP in a language independent manner. Also it was proved that the design patterns could be used in a platform independent manner.

Future subjects include refinement of the models specified in this paper, and specification of other patterns.

REFERENCES

Chiba, S., 2005, *Aspect-Oriented Programing*, Gijutsu-Hyohron Co., Ltd, Tokyo, Japan.

Gamma, E. et al, 1995, *Design Patterns*, Addison-Wesley Publishers, Indiana, USA.

Hannemann, J. and Kiczales, G., 2002, Design Pattern Implementation in Java and AspectJ, *Proc. of ACM OOPSLA*, pp 161-173.

Kennedy Carter Ltd., *"iUML"*, http://www.kc.com/.

Mellor, S. J. and Balcer, M. J., 2002 *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley Publishers, Indiana, USA.

Mukasa, H. et al, 2007, Design Patterns in Executable UML, *Proc. IPSJ/IEICE Forum on Information Technology*, Vol.4, pp.437-438.

Nagase, Y. et al, 2004, *Introduction to Aspect-Oriented programming*, Soft Bank Publishing Ltd, Tokyo, Japan.

OMG, 2000, *"UML2.0"*, http://www.uml.org/.

OMG, 2001, *"UML Action Semantics"*, http://www.omg.org/cgi-bin/doc?ptc/02-01-09.

Teruya, A. et al, 2008, Embedded System Design Based on Aspect-Oriented Executable UML, *Proceedings of 8th International Conference on Applied Computer Science*, pp 247-252.