

# EVOLVING DESIGN PATTERNS BASED ON MODEL TRANSFORMATION

Jing Dong, Sheng Yang, Dung T. Huynh  
Department of Computer Science  
University of Texas at Dallas  
Richardson, TX 75083, USA  
{jdong,syang,huynh}@utdallas.edu

## ABSTRACT

In this paper, we propose two-level transformations to capture the evolution processes of design patterns, which are generally implicit in the descriptions of design patterns. These two-level transformations are the primitive-level and pattern-level evolutions. The evolution processes are implemented based on XML Metadata Interchange (XMI) format to transform the UML models of design pattern applications.

## KEYWORDS

Design pattern, Model Transformation, XMI, XSLT, Design pattern evolution

## 1. Introduction

Change is a constant theme of software design and development. Due to constant changes of user requirements, platforms, technologies and environments, software systems need to be adapted to such changes. Unlike other engineering products, such as automobiles and electronic devices, software systems are normally more amenable to changes. It can be a disaster if a single change may cause a large number of changes in the software systems. It is important to localize the changes such that minimum efforts are needed. This requires the initial designers of a software system to be aware of potential changes. Thus, the resulting software systems are flexible and agile to future evolution.

Design patterns [5] capture expert design experience by partitioning software designs into stable part and changeable part. By separating and encapsulating both parts, the change impact of a software design can be minimized. One of the important goals of design patterns is design for change. Thus, most of design patterns encapsulate future changes that may only affect limited part of a design pattern. This evolution process can be achieved by adding or removing design elements in existing design patterns. In the document of each design pattern, however, the evolution information is generally not explicitly specified. When changes are needed, a designer has to read between the lines of the document of a design pattern to figure out the correct ways of changing the design. More importantly, the evolution process of a design pattern may involve the addition or removal of several parts of a design pattern.

Misunderstanding of a design pattern may result in missing parts of the evolution process. The addition and removal of system parts should not violate the constraints and properties of design patterns. Thus, it is important to have, in the documentation of the design pattern, information about the evolution of the patterns. The evolution of a software system at the design level is less costly than it is at the implementation level.

Design patterns are usually represented in the Unified Modeling Language (UML) [2] which is considered to be the *de facto* standard for object-oriented modeling. The Model Driven Architecture (MDA) [8] supports developing software systems based on models as primary artifacts. Thus, the level of abstraction of software development is raised from implementation (writing code) to model transformation. By raising the level of abstraction, the level of reuse is raised accordingly since high-level software models can be reused as well as software programs (libraries). In this way, models become assets in MDA. Consequently, technology that supports the transformation of models is considered as a key enabler of MDA. While the application of a design pattern can be represented in a design model in UML, the evolution of the design pattern may be considered as a transformation of the design model.

The XML Metadata Interchange (XMI) [9] is an interchange format for metadata in terms of the Meta Object Facility (MOF) [8]. XMI specifies how UML models are mapped into a XML file. By representing a UML models in XML format, the UML model can be manipulated since there are rich collections of XML related techniques and tools available. The extensible stylesheet language transformation (XSLT) [10] provides the transformation from XML document to other types of document (including XML). The use of XMI and XSLT helps on the automated model transformation process and enforces constraints of model implicitly.

In this paper, we propose two-level transformations, the primitive level and the pattern level, to capture the evolution processes of design patterns. The primitive-level transformations are the addition or removal of modeling elements, such as classes and relationships. The pattern-level transformations characterize the recurring evolutions of each design pattern based on the primitive-level transformations. Meanwhile, the evolution processes are implemented based on XML Metadata Interchange

(XMI) format to transform the UML models of design pattern applications. Thus, the structure of a design pattern may be evolved in some prescribe ways based on the pattern-level transformation.

The reminder of this paper is organized as follows: the next section describes the primitive-level and pattern-level evolutions. Section 3 presents the transformations of UML models of design patterns based on XMI. The last two sections are related work and conclusions.

## 2. Classifications of pattern evolutions

One of the important goals of design patterns is design for change since change is a constant theme in software development. A design pattern normally encapsulates some particular ways for future changes. After a design pattern is applied in a software application, the designer may change the application design in the particular ways directed by the design pattern. The evolution information of each design pattern allows the designers to change the system design with minimum impact of other part of the system. For example, Figure 1 shows the class diagram of the Mediator pattern. When the Mediator pattern is applied initially, there may be only two concrete colleague classes. A change may request additional concrete colleague class later as shown in Figure 2. However, such evolution information of each design pattern is normally implicit to the description of the pattern. A designer has to search the document of the

pattern to find the guidance on evolution. Misunderstanding and mistakes of changing the design patterns may compromise the benefits of using these patterns and have huge impact on the system designs.

In this section, we investigate different kinds of evolutions in design patterns and provide a classification of these evolutions. We describe these evolutions in terms of two-level transformations: the primitive-level evolution and the pattern-level evolution.

The primitive-level evolution describes the basic transformations that can be performed during the evolution process of a design pattern. These basic transformations include the addition or removal of a modeling element, such as class, operation, attribute, association, generalization, aggregation, composition, realization, and dependency. These basic transformations become the building blocks of the pattern-level evolution.

The pattern-level evolution characterizes the recurring evolution processes which occur in many design patterns. It is described in terms of a sequence of the basic transformations. Each design pattern may perform some of the pattern-level evolutions, which can be added in the document of the pattern. Thus, the designer may choose a potential pattern-level evolution and apply the corresponding transformations when changes are required.

**Table 1 The Primitive-Level Evolutions**

Model Elements	Parameter List	Descriptions
Class	className	Add or remove a class with name “className” into a pattern
Attribute	attributeName, className, type, accessibility	Add or remove an attribute with name “attributeName”, type of “type”, accessibility of “accessibility” into the class “className”
Operation	operationName, className, returnType, accessibility, para1, paraType1...	Add or remove an operation with name “operationName”, type of “type”, accessibility of “accessibility”, and arguments list para1 with type “paraType1” into the class “className”
Association	className1, className2	Add or remove an association between classes “className1” and “className2” into a pattern
Generalization	child, parent	Add or remove a generalization relationship into a pattern, with subclass “child” and superclass “parent”
Aggregation	part, whole	Add or remove an aggregation relationship into a pattern, “part” class is a part of “whole” class
Composition	part, whole	Add or remove a composition relationship into a pattern, “part” class is a part of “whole” class
Realization	fromName, toName	Add or remove a realization relationship from class “fromName” to class “toName” into a pattern
Dependency	fromName, toName	Add or remove a dependency relationship from class “fromName” to class “toName” into a pattern

### 2.1 Primitive-Level Evolutions

We identify nine modeling elements that can be added or deleted as the basic transformations in the pattern evolution processes. The general format of adding a modeling element is Add (ME (PL)). The model elements (ME) and the parameter list (PL) are shown in Table 1.

For example, adding a class named “Leaf” can be specified: Add (Class (Leaf)). Similarly, the removal of a modeling element can be specified: Delete (ME (PL)). The replacement of a model element with another is conducted by first removing the modeling element and then adding a new modeling element. It can be defined: Delete (ME1 (PL1)) + Add (ME2 (PL2)).

## 2.2 Pattern-Level Evolutions

In this section, we characterize several pattern-level evolutions which are recurring in different design patterns. Each design pattern may encapsulate some of the pattern-level evolutions, which can be explicitly documented in the descriptions of the pattern. Thus, a designer can simply follow the prescribed evolution processes when the corresponding changes are needed. Note that we do not claim this is a complete list of all possible pattern-level evolutions. Nevertheless, new pattern-level evolutions can be easily added into the list specified by the primitive-level evolutions.

The first pattern-level evolution is a simple addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern. This class is independent in the sense that the addition or removal of the class does not cause any effects on the existing classes of the design. This kind of pattern-level evolution can be expressed in the primitive level evolutions as follows<sup>1</sup>:

```
Add ( Class (className)) +
Add ( Relationship (className, existingClassName))
```

where className is the name of the class which is added into the pattern. Relationship includes association, generalization, aggregation, composition, realization, and dependency. The existingClassName is the name of the class from the original pattern. There may be multiple relationships added into the pattern with the addition of a class.

This kind of evolution appears in several design patterns as, for example, in the Mediator and Facade patterns. Figure 1 is the class diagram of the Mediator pattern describing a possible application containing two ConcreteColleague classes. A potential evolution of this pattern application is to add a new ConcreteColleague class, which can be defined as the following transformation in terms of the primitive-level evolutions:

```
Add ( Class (ConcreteColleague)) +
Add ( Generalization (ConcreteColleague, Colleague)) +
Add ( Dependency (ConcreteMediator, ConcreteColleague))
```

where a new ConcreteColleague class is added with two new relationships: generalization and dependency. The generalization relationship is with the Colleague class. The dependency relationship is on the ConcreteMediator class. The result of this evolution is shown in Figure 2.

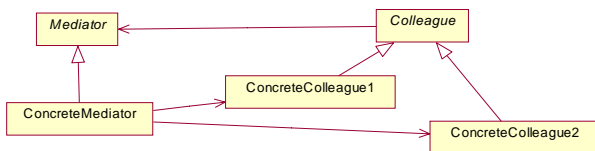


Figure 1 Mediator Pattern with Two Concrete Colleagues

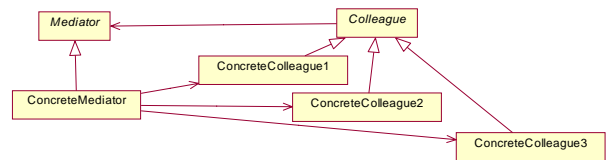


Figure 2 Mediator Pattern with Three Concrete Colleagues

The second pattern-level evolution is the addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern. In addition, certain attributes and/or operations of this class are added and removed accordingly. This kind of pattern-level evolution can be expressed in the primitive level evolutions as follows:

```
Add ( Class (className)) +
Add ( Relationship (className, existingClassName)) +
Add ( Attribute (attributeName, className, type, accessibility)) +
Add ( Operation (operationName, className, returnType, accessibility,...))
```

where className is the name of the class which is added into the pattern. Relationship includes association, generalization, aggregation, composition, realization, and dependency. The existingClassName is the name of the class from the original pattern. The attributeName is the name of the attribute of the class to be added whereas the operationName is the name of the operation of the class. There may be multiple relationships, attributes and/or operations added into the pattern with the addition of a class.

This kind of evolution can be found in several design patterns as, for example, in the Composite, Bridge, State, Strategy, Chain of Responsibility, and Observer patterns. Figure 3 is the class diagram of the Observer pattern describing a possible application containing one ConcreteSubject and two ConcreteObserver classes. A potential evolution of this pattern application is to add a new ConcreteObserver class (ConcreteObserver3) with its attributes (s1 and s2) as shown in Figure 4, which can be defined as the following transformation in terms of the primitive-level evolutions:

```
Add ( Class (ConcreteObserver3)) +
Add ( Generalization (ConcreteObserver3, Observer)) +
Add ( Attribute (s1, ConcreteObserver3, Undefined, private)) +
Add ( Attribute (s2, ConcreteObserver3, Undefined, private))
```

where a new concrete observer (ConcreteObserver3) is added with a generalization relationship between this new class and the Observer class. Two attributes (s1 and s2) of this new class are also added accordingly, where “Undefined” refers to the undefined types of these two attributes.

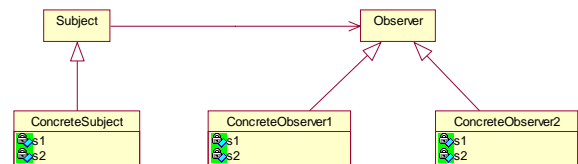


Figure 3 Observer Pattern with Two Concrete Observers

<sup>1</sup> Since the addition and removal have the same format and the only difference is the transformation names (Add and Delete), we omit the evolutions of removing modeling elements.

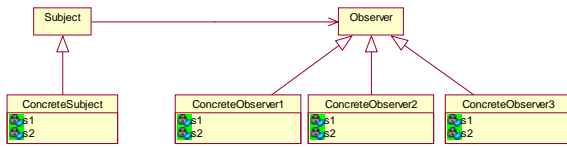


Figure 4 Observer Pattern with Three Concrete Observers

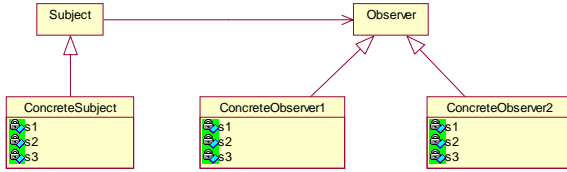


Figure 5 Observer Pattern with Three Attributes

The third kind of pattern-level evolution is the addition or removal of one attribute/operation in several different classes consistently. In this case, a certain set of classes, instead of a single class, are affected by the addition or removal of the attribute or operation. This kind of pattern-level evolution can be expressed in the primitive level evolutions as follows:

$$\sum_i \text{Add} (\text{Attribute}(\text{attributeName}_i, \text{className}_i, \text{type}_i, \text{accessibility}_i)) +$$

$$\sum_j \text{Add}(\text{Operation}(\text{operationName}_j, \text{className}_j, \text{returnType}_j, \text{accessibility}_j, \dots))$$

where attributeName is the name of the attribute of the adding class named className<sub>i</sub>; whereas operationName is the name of the operation of the adding class named className<sub>j</sub>. There may be multiple relationships, attributes and/or operations added into the pattern with the addition of a class.

This kind of evolution is common in several design patterns as, for example, in the Decorator and Observer patterns. Figure 3, for instance, shows an application of the Observer pattern with one ConcreteSubject and two ConcreteObserver classes. One potential evolution is to add one attribute called s3 as a new data to be observed by the observers. Thus, this attribute needs to be added in all ConcreteSubject and ConcreteObserver classes, which can be defined as the following transformation in terms of the primitive-level evolutions:

$$\text{Add} (\text{Attribute}(s3, \text{ConcreteSubject}, \text{Undefined}, \text{private})) +$$

$$\text{Add} (\text{Attribute}(s3, \text{ConcreteObserver1}, \text{Undefined}, \text{private})) +$$

$$\text{Add} (\text{Attribute}(s3, \text{ConcreteObserver2}, \text{Undefined}, \text{private}))$$

which indicates the attribute s3 is added into the ConcreteSubject, ConcreteObserver1, and ConcreteObserver2 classes. The resulting class diagram of this evolution is shown in Figure 5.

The fourth kind of pattern-level evolution is the addition or removal of a group of correlated classes. When certain classes are added or removed, some other classes have to be added or removed accordingly. These correspondence relations are important since missing transformations may cause inconsistency. In addition, the corresponding relationships between this group of classes and other classes are added or removed. The attributes and operations of this group of classes are also added or removed. The addition or removal of this group of classes may not affect the internal of other classes in the original

design pattern applications. This kind of pattern-level evolution can be expressed in the primitive level evolutions as follows:

$$\sum_i \text{Add} (\text{Class}(\text{className}_i)) +$$

$$\sum_{ij} \text{Add} (\text{Relationship}(\text{className}_i, \text{className}_j)) +$$

$$\sum_{ij} \text{Add} (\text{Attribute}(\text{attributeName}_i, \text{className}_j, \text{type}_i, \text{accessibility}_i)) +$$

$$\sum_{ij} \text{Add} (\text{Operation}(\text{operationName}_i, \text{className}_j, \text{returnType}_i, \text{accessibility}_i, \dots))$$

where attributeName<sub>i</sub> is the name of the i<sup>th</sup> attribute of the adding class named className<sub>i</sub>; whereas operationName<sub>i</sub> is the name of the i<sup>th</sup> operation of the added class named className<sub>j</sub>. There may be multiple relationship, attributes and/or operations added into the pattern with the addition of a class.

This kind of evolution can be seen in several design patterns as, for example, in the Builder, Factory Method, Command, Interpreter, Iterator, Visitor, Abstract Factory patterns. Figure 6 shows an application of the Abstract Factory pattern with two kinds of products (AbstractProductA and AbstractProductB). Each kind of products has two concrete products: ProductA1/ProductB1 and ProductA2/ProductB2, respectively. Thus, there are two concrete factories: ConcreteFactory1 and ConcreteFactory2. A potential evolution can be the addition of a new kind of concrete products (ProductA3 and ProductB3). This requires the addition of a new concrete factory (ConcreteFactory3) to create the corresponding newly added concrete products. This new concrete factory class also has the same operations (createProductA and createProductB) as the other two concrete factory classes as shown in Figure 7. This evolution can be defined in terms of the primitive-level transformations as follows:

$$\text{Add} (\text{Class} (\text{ConcreteFactory3})) +$$

$$\text{Add} (\text{Class} (\text{ProductA3})) +$$

$$\text{Add} (\text{Class} (\text{ProductB3})) +$$

$$\text{Add} (\text{Generalization} (\text{ProductA3}, \text{AbstractProductA})) +$$

$$\text{Add} (\text{Generalization} (\text{ProductB3}, \text{AbstractProductB})) +$$

$$\text{Add} (\text{Generalization} (\text{ConcreteFactory3}, \text{AbstractFactory})) +$$

$$\text{Add} (\text{Realization} (\text{ConcreteFactory3}, \text{ProductA3})) +$$

$$\text{Add} (\text{Realization} (\text{ConcreteFactory3}, \text{ProductB3})) +$$

$$\text{Add} (\text{Operation} (\text{createProductA}, \text{ConcreteFactory3}, \text{null}, \text{public})) +$$

$$\text{Add} (\text{Operation} (\text{createProductB}, \text{ConcreteFactory3}, \text{null}, \text{public}))$$

The fifth kind of pattern-level evolution is the addition or removal of a group of classes. This change also requires the addition or removal of some attributes or operations in the classes of the original pattern applications.

The expression of this kind of pattern-level evolution is the same as the one of the fourth kind pattern-level evolution. Nevertheless, they have different semantic meaning. In the fourth kind of pattern-level evolution, the className<sub>j</sub> in the “Add” attributes and operations transformations only includes those classes which are newly added into the pattern. In the fifth kind of pattern-level evolution, in contrast, className<sub>j</sub> includes the newly added classes and existing classes, i.e., the addition of a group of classes results in the addition of the attributes and operations of the existing classes in original pattern.

This kind of evolution can be seen in several design patterns as, for example, in the Abstract Factory and Adapter patterns. For the same example shown in Figure 6, another potential evolution can be the addition of a new kind of product (AbstractProductC with ProductC1 and ProductC2). This requires the addition of the createProductC operation in all concrete factory classes (ConcreteFactory1 and ConcreteFactory2). The corresponding generalization and realization relationships are also added as shown in Figure 8. This evolution can be defined in terms of the primitive-level transformations as follows:

```

Add ( Class (AbstractProductC)) +
Add ( Class (ProductC1)) +
Add ( Class (ProductC2)) +
Add ( Generalization (ProductC1, AbstractProductC)) +
Add ( Generalization (ProductC2, AbstractProductC)) +
Add ( Realization (ConcreteFactory1, ProductC1)) +
Add ( Realization (ConcreteFactory2, ProductC2)) +
Add ( Operation (createProductC, ConcreteFactory1, null, public)) +
Add ( Operation (createProductC, ConcreteFactory2, null, public))

```

which indicates that three classes, AbstractProductC, ProductC1, and ProductC2 are added into the pattern application. ProductC1 and ProductC2 are subclasses of AbstractProductC. The createProductC() operation is added into both ConcreteFactory1 and ConcreteFactory2 classes. The realization relationships are also added between the ConcreteFactory1 and ProductC1 classes and between the ConcreteFactory2 and ProductC2 classes, respectively. These relationships show that ConcreteFactory1 and ConcreteFactory2 create ProductC1 and ProductC2, respectively.

### 3 Automating the Pattern Evolution

In the previous section, we introduce two levels of evolutions: primitive level and pattern level. We also explicitly describe the evolution processes of several design patterns in terms of the two-level transformations. In order to automate the evolution process, we use XMI to describe our two-level evolutions and XSLT to declare the transformation rules in this section. Using an XSLT processor, design pattern evolutions can be automated by transforming from the original UML model of a design pattern to the destination UML model of the pattern.

In Section 2.1, we present the primitive-level evolutions which can be implemented in XMI. More specifically, the addition or removal of a modeling element can be implemented in XMI by the tags <XMI.Add> Subtags </XMI.Add> and <XMI.Delete> Subtags </XMI.Delete>, respectively. Table 2 shows the subtags corresponding to the primitive-level evolutions in Table 1, respectively. For instance, the first primitive-level evolution Add(Class(className)) can be implemented in XMI as follows:

```

<XMI.Add>
  <UML:Class name = “...” />
</XMI.Add>

```

The pattern-level evolutions are implemented in XMI similarly. For example, the second kind of pattern-level evolution (Section 2.2) can be implemented in XMI as shown in Figure 10.

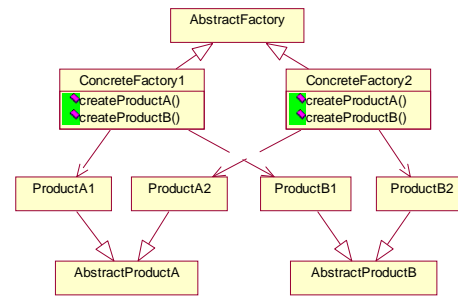


Figure 6 Abstract Factory Pattern with Two Kinds of Products Created by Two Concrete Factories

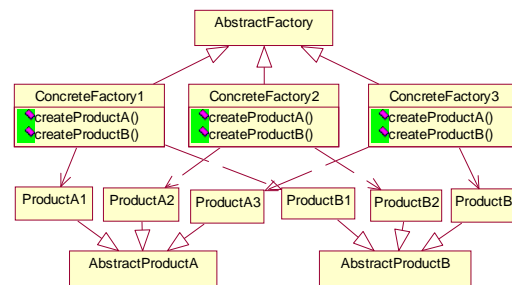


Figure 7 Abstract Factory Pattern with Three Kinds of Concrete Products Created by Three Concrete Factories

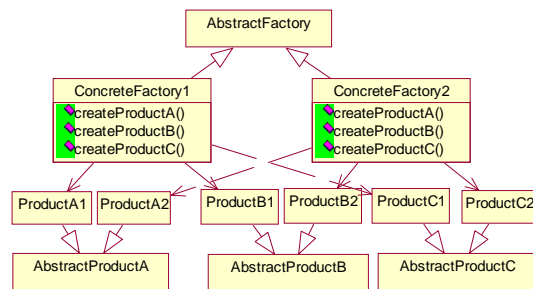


Figure 8 Abstract Factory Pattern with Three Kinds of Products Created by Two Concrete Factories

Figure 9 depicts the architecture of our proposed solution for automating pattern evolutions. The transformation program is an XSLT processor which can automatically transform the UML model of a pattern based on its inputs. There are three inputs to the transformation program. The first one is the original UML model, which has been converted into XMI format. It records all the model elements, such as classes, attributes, operations, and relationships, in the original UML model of a pattern. This file can be generated by Rational Rose [11] Plug-in tool called UniSys. The second input includes all model elements to be added or removed in/from the UML model. This file contains a pattern-level evolution (Section 2.2) in XMI format. The third input file is the transformation rules, which is written in XSLT. This file tells the program where these new modeling elements should be added or removed. The

output of the program is the evolved UML model of a pattern in XMI format. This file can be imported by Rational Rose and transformed into a UML class diagram.

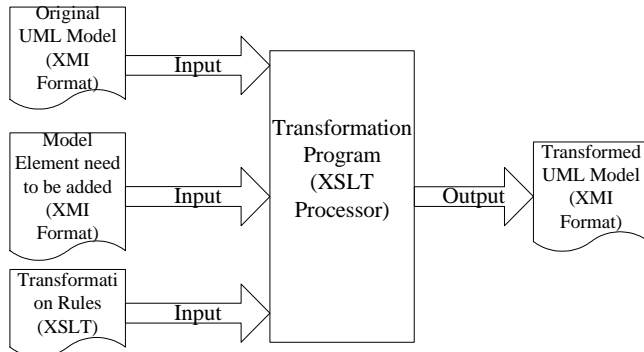


Figure 9 The Solution for Automating Pattern Evolutions

The second pattern-level evolution is expressed in XMI shown in Figure 10, where a new class named “ConcreteObserver3” is added into the original

application of the Observer pattern. The two attributes: s1 and s2 of the ConcreteObserver3 class are also added. In addition, the generalization relationship between the ConcreteObserver3 and Observer classes are added with the ConcreteObserver3 class as child and the Observer class as parent. The transformation program merges these two files together based on the transformation rules defined in XSLT file and produces an evolution of the application of the Observer Pattern with three concrete observers (shown in Figure 4).

```
<XMI.Add>
  <UML:Class name="ConcreteObserver3" />
</XMI.Add>
<XMI.Add>
  <UML:Attribute attributeName="s1" className="ConcreteObserver3" />
</XMI.Add>
<XMI.Add>
  <UML:Attribute attributeName="s1" className="ConcreteObserver3" />
</XMI.Add>
<XMI.Add>
  <UML:Generalization child="ConcreteObserver3" parent="Observer" />
</XMI.Add>
```

Figure 10 Pattern-Level Evolution of the Observer Pattern in XMI

Table 2 The Primitive-Level Evolutions in XMI

Evolution	Subtags of <XMI.Add> and <XMI.Delete>
class	<UML:Class name = “...” />
attribute	<UML:Attribute attributeName = “...” className = “...” />
operation	<UML:Operation operationName = “...” className = “...” />
association	<UML:Association associationEnd1 = “...”associationEnd2 = “...” />
generalization	<UML:Generalization child = “...” parent = “...” />
aggregation	<UML:Aggregation wholeName = “...” partName = “...” />
composition	<UML:Composition wholeName = “...” partName = “...” />
realization	<UML:Realization fromName = “...” toName = “...” />
dependency	<UML:Dependency fromName = “...” toName = “...” />

## 4 Related Work

The evolution processes of design patterns have been studied in [1], where Prolog [3] is used to capture the structural evolution processes of design patterns. The structural aspect of a design pattern is described in terms of Prolog facts. Thus, the evolution and change of a design pattern application can be achieved by the addition or removal of new or old Prolog facts. The evolution processes are defined as Prolog rules. In this paper, we characterize two-level evolutions and implement the model transformations based on XMI and XSLT.

A generic XMI-based transformation infrastructure of UML models has been presented in [7]. This allows the user to select a predefined generic XML-based transformation and configure its parameters. Unlike this work, we concentrate on the XMI-based transformations for design pattern evolutions.

The tool support for UML model evolution is provided in [6], which is also based on XMI. The design and development of the tool applies several design

patterns. In contrast, we focus on the evolution of design patterns, instead of the evolution of UML models.

Experiments have been conducted in [4] to show that XMI can be used to transform the UML models into other modeling languages, such as SQL. The implementation of the XMI-based transformation uses XSLT. We base on these experiments and use XSLT to implement the transformations.

## 5 Conclusion

Currently, the evolution information of each design pattern is generally implicit in the descriptions of the pattern. A designer has to dig into the pattern descriptions and try to understand the particular ways of evolutions encapsulated in the design patterns. There are several problems when the evolution information is implicit: first, it is hard for the designer to take advantage of the benefits of using a design pattern when changes are needed. Second, the evolution of a design pattern generally involves several classes and relationships. Missing one part may cause inconsistencies and errors in the design

which are difficult to fine and correct. Third, the evolution processes are not reusable if not documented. As discussed previously, many of the evolution processes recurs in different patterns.

In this paper, we characterize two-level transformations: the primitive level and the pattern level and explicitly describe the evolution of design patterns using these two-level transformations. We also implement the transformation based on XMI. In the future, we will characterize the constraints of evolutions of each design pattern and provide techniques and tools for checking such constraints after evolutions.

As another future work, we are investigating the model transformation techniques based on Query, View, Transformation (QVT) that is a forthcoming OMG standard allowing users to query, establish and maintain views, and transform MOF models. Many groups have submitted their proposals and they are still competing. There are also model transformation tools available, such as Model Transformation Framework from IBM [12] of which we can take advantage. When the QVT is standardized by OMG, we will apply the QVT techniques for the transformation of design patterns.

## References

- [1] P. Alencar, D. Cowan, J. Dong, and C. Lucena, A Pattern-Based Approach to Structural Design Composition, the Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference (COMPSAC), pp160-165, Phoenix USA, October 1999.
- [2] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide, Addison-Wesley, 1999.
- [3] W. F. Clocksin and C.S. Mellish. Programming in Prolog. Berlin : Springer-Verlag, 1987.
- [4] B. Demoth, H. Hussmann, and S. Obermaier. "Experiments with XM-based Transformations of Software Models", *Workshop on Transformations in UML (ETAPS 2001 Satellite Event)*, Genova, Apr. 2001
- [5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994
- [6] F. Keienburg and A. Rausch, "Using XML/XMI for tool Supported Evolution of UML Models", *Proceeding of International Conference Hawaii International Conference on System Science (HICSS)*, Maui, Jan. 2001, IEEE 2001
- [7] J. Kovse and T. Harder, "Generic XMI-Based UML Model Transformations", *Proceedings of the International Conference on Object-Oriented Information Systems (OOIS)*, Montpellier, Sept. 2002, Springer-Verlag, pp. 192-198.
- [8] Model Driven Architecture. <http://www.omg.org/mda/>
- [9] W3C, Extensible Markup Language (XML), <http://www.w3.org/>
- [10] W3C, XSL Transformations (XSLT), <http://www.w3.org/>
- [11] Rational Rose website. <http://www.rational.com/>
- [12] IBM, [http://www-128.ibm.com/developerworks/rational/library/05/503\\_sebas/](http://www-128.ibm.com/developerworks/rational/library/05/503_sebas/)