

# QVT BASED MODEL TRANSFORMATION FOR DESIGN PATTERN EVOLUTIONS

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

Yongtao Sun  
*American Airlines  
4333 Amon Carter Blvd  
Fort Worth, TX 76155, USA  
Yongtao.Sun@aa.com*

W. Eric Wong  
*Computer Science Department  
University of Texas at Dallas  
Richardson, TX 75083, USA  
ewong@utdallas.edu*

## ABSTRACT

In this paper, we present our methods on explicitly documenting the evolution processes of a design pattern based on two-level transformations. We provide tool support to automate such process based on the Query, View, Transformation (QVT). In this way, a software system design with the applications of design patterns represented in UML model may be automatically evolved to a new design based on the model transformation rules defined for the corresponding design patterns applied.

## KEY WORDS

Design pattern, Model Transformation, UML, QVT, MDA, Eclipse, Design pattern evolution.

## 1 Introduction

Large web-based applications become more and more difficult to design due to increasing size and complexity. To cope with the complexity, design patterns [6] have been proposed to reuse good design experience in different application domains, such as hypermedia design patterns [13]. Design patterns capture expert design experience by partitioning software designs into a stable part and changeable part. By separating and encapsulating both parts, the change impact of a software design can be minimized. Thus, most of the design patterns encapsulate future changes that may only affect a limited part of a design pattern. This evolution process can be achieved by adding or removing design elements in existing design patterns. In the documentation 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 documentation 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.

The Query, View, Transformation (QVT) [18] is a specification by the Object Management Group (OMG). The QVT standardizes the model transformation by splitting model transformations into two-level architecture: relations metamodel and core metamodel. In the relations metamodel, the model transformation between model candidates is specified as a set of relations, as for example illustrated in Figure 1. These relations must hold for a successful model transformation. The core metamodel is defined using minimal extension to essential MOF (EMOF) and OCL. A transformation in the core metamodel is defined as a set of mapping. The relations metamodel can be transformed to the core metamodel by a set of mapping rules.

In this paper, we explicitly define the evolutions of design patterns as model transformation rules based on two-level evolution processes: the primitive level and the pattern level. These model transformation rules are defined in QVT which may transform a design model to a new model by adding or removing modeling elements. For each design pattern, thus, we define the model transformation rules in QVT for its possible evolutions. We also automate such evolution processes in the IBM Eclipse environment.

The remainder of this paper is organized as follows: the next section introduces the transformation rules of design pattern evolutions based on QVT. Section 3 discusses our tool support for automating the model transformation processes. The last two sections cover related work and conclusions.

## 2. QVT Based Model Transformation

In this paper, we investigate the QVT-based model transformations for the pattern-level evolutions. Unlike automating the pattern evolution by XSLT processor [5], the pattern evolutions based on IBM Model Transformation Framework (MTF) perform the evolution in a single step, which reduces execution time and improves the evolution performance. MTF is a set of tools which help designers to develop transformations between the Eclipse Modeling Framework (EMF) models, such as the UML2, Java Development Tools (JDT), XML Schema Infoset Model (XSD), and Ecore models. The Ecore model is an essential model in EMF and all other models in EMF are mapped to Ecore either explicitly or

implicitly. Model transformation in EMF is performed by creating mapping objects between source model and target model. Transformation in EMF is defined in a declarative way, i.e., the user can specify a set of transformation rules between models, and the transformation engine performs the transformation between models based on these pre-defined transformation rules. The transformation engine performs model transformation in two steps, mapping and reconciliation. In the mapping stage, the transformation engine evaluates the transformation rules and generates the mapping between source model objects and target model objects. By performing mapping between source and target model objects, some models may become inconsistent with respect to the pre-defined transformation rules. The reconciliation stage tries to solve these inconsistencies by creating missing elements, modifying existing elements and deleting elements.

The transformation rules are defined in the EMF as a set of relations within one file. Figure 1 illustrates an example of the relation between two classifiers (ClfToClf), which maps one classifier (e.g., class or interface) to another classifier depending on the type of the model elements defined for design pattern evolution. The relation ClfToClf (line 1 and 2 in Figure 1) has two arguments, the source classifier (srcClf) and the target classifier (tgtClf), which represents the source class/interface and the target class/interface of the transformation, respectively. Both arguments have a type of “uml:Classifier” where the prefix “uml” is the namespace which is associated with the EMF Ecore model at <http://com.ibm/mtf/uml.ecore> and the “Classifier” is one of the types defined in the uml Ecore model. The relation ClfToClf is defined as *abstract* because we are dealing with two different types of classifier in a single mapping relation. Depending on the types of the arguments in the relation, a different type of classifier mapping is performed. If the arguments have the type of “uml:Class”, for instance, the mapping relation ClassToClass (lines 3 to 8 in Figure 1) is performed to map the source class (srcClass) in the source model to the target class (tgtClass) in the target model. The *when* clause in the relation ClfToClf states that the mapping will perform only if the source classifier and the target classifier have the same name.

The mapping relation ClassToClass is shown from lines 3 to 8 in Figure 1, which is a sub-relation of relation ClfToClf to deal with the mapping between classes from the source model and classes from the target model. In the body of the ClassToClass relation, the relation hierarchically maps the subclasses of the class (line 4), the generalization relationship the class has (line 5), the attributes the class has (line 6), and the operations the class has (line 7). The relation ClfToClf in line 4 is invoked in relation ClassToClass to map the subclasses of the source class by recursively invoking the mapping relation ClfToClf with the nestedClassifier of the source class as arguments. The keyword *over* in relation

ClfToClf invocation in line 4 indicates that the corresponding argument represents collections and a mapping should be created between elements of the collections. Hence, the rule of line 4 simply means to create mappings between all subclasses of *srcClass* and all subclasses of *tgtClass*. The invocation of relation GenToGen in line 5 maps the generalization relationship of the mapped class. The keyword *ordered* in the rule of line 5 indicates that the mappings between the elements which appear in the same positions within their respective domains are created. The multiplicity [0..1] in generalization mapping relation means that the mapping between the generalization relationship should be mapped at most once. In addition, the relation PropToProp in line 6 and the relation OpToOp in line 7 map all attributes and operations between the classes from source model and the classes from target model, respectively.

Similarly, the relation IntToInt (lines 9 through 14) 7 in Figure 1 defines the mapping relation from the interfaces of the source model to the interfaces of the target model. It also hierarchically maps the sub-interfaces, generalization relationship, attributes and operations of the mapped interface.

```

1) abstract relate ClfToClf(uml:Classifier srcClf, uml:Classifier tgtClf)
2)   when equals(srcClf.name, tgtClf.name)

3) relate ClassToClass extends ClfToClf (uml:Class srcClass, uml:Class tgtClass) {
4)   ClfToClf(over srcClass.nestedClassifier, over tgtClass.nestedClassifier),
5)   ordered GenToGen [0..1](over srcClass.generalization, over tgtClass.generalization),
6)   PropToProp(over srcClass.ownedAttribute, over tgtClass.ownedAttribute),
7)   OpToOp(over srcClass.ownedOperation, over tgtClass.ownedOperation)
8) }

9) relate IntToInt extends ClfToClf (uml:Interface srcInt, uml:Interface tgtInt) {
10)  ClfToClf(over srcInt.nestedClassifier, over tgtInt.nestedClassifier),
11)  ordered GenToGen (over srcInt.generalization, over tgtInt.generalization),
12)  PropToProp(over srcInt.ownedAttribute, over tgtInt.ownedAttribute)
13)  OpToOp(over srcInt.ownedOperation, over tgtInt.ownedOperation)
14) }

```

Figure 1 Mapping Rule in MTF

### 3. Automating the Pattern Evolution by Model Transformation

There are typically three kinds of actions that can be taken to perform the design pattern evolutions. The first action is to add a model element or a group of model elements, such as classes, attributes/operations, and relationships into the original system design. The second action is to remove model element(s) from the original system design. The third action is to replace existing model elements by some new model elements. Since replacing existing model elements can be considered as first removing the existing model elements from the system design, then adding new model elements to the system design, we only discuss the addition and removal of model elements in the following sections.

#### 3.1 The Addition of Model Elements

Adding a model element into a system may result in the addition of a group of model elements into the system depending on the type of pattern-level evolution the user chooses. For instance, Figure 2 shows an Observer pattern with two concrete observer classes,

ConcreteObserver1 and ConcreteObserver2. There are two states, s1 and s2 in classes ConcreteObserver1, ConcreteObserver2, and ConcreteSubject. There exist two kinds of pattern-level evolutions for the Observer pattern [4][5]. First, we can add a new class, say ConcreteObserver3, which contains two states, s1 and s2, into the Observer pattern. In addition, a generalization relationship between class ConcreteObserver3 and class Observer needs to be added into the Observer pattern. Figure 3 shows the evolved Observer pattern with three concrete observer classes. Second, we can add a state, say s3, to class ConcreteSubject, state s3 is also added into classes ConcreteObserver1 and ConcreteObserver2 to keep the observer pattern consistent. Figure 4 shows the evolved Observer pattern with three states, s1, s2 and s3.

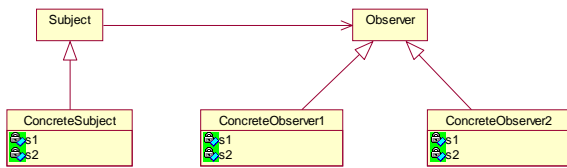


Figure 2 Observer Pattern with Two Concrete Observers

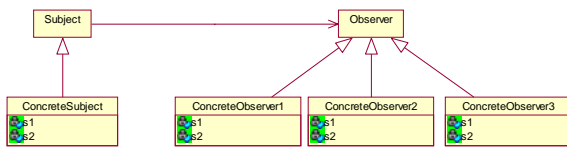


Figure 3 Observer Pattern with Three Concrete Observers

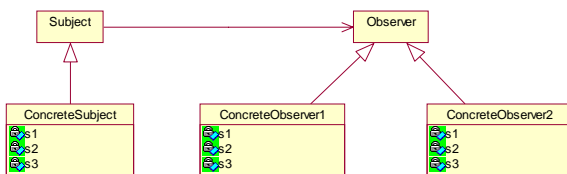


Figure 4 Observer Pattern with Three Attributes

When the design patterns are applied and evolved, the names of the classes, attributes, and operations may not be the same as the role names they play in the pattern. For example, the new concrete observer class may not be named “ConcreteObserver3” in some applications. It may be named, for example, “ChartView”. Hence, we provide an assistant tool for the designer to supply the names of the modeling elements that are added in the design. Our tool can automatically add the corresponding modeling elements into the original design based on the pattern-level evolution process of the corresponding pattern, e.g., the *packaged* pattern-level evolution [4] of the Observer pattern in this case.

Figure 5 illustrates the user interface for the designer to perform pattern evolution. After the Observer pattern expressed in XMI-UML2 format is uploaded by the user interface, the design pattern applied in the system and the possible evolutions for each design pattern are shown in the left panel of the interface in a tree structure. For this example, there exists only the Observer pattern. Thus, only Observer pattern is shown. The items under each design pattern are possible pattern-level evolutions for

that design pattern, e.g., for the applications of the Observer pattern, there are two possible evolutions, adding/removing a concrete observer class and adding/removing a state. Each possible evolution has two actions, *Add* and *Remove*. When the designer selects the *Add* action in the Observer pattern, for example shown in Figure 5, the right panel will display a table to collect all necessary information to perform the *Add* action. The user enters the name of concrete observer and click on the *Add* button to perform the addition of a concrete observer class into the Observer pattern.

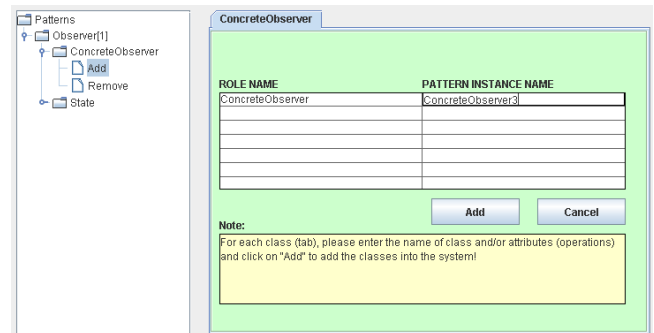


Figure 5 User Interface for Adding Model Element

Figure 7 explains the process when the *Add* button is clicked in the user interface. The flow labeled with the plus (+) sign indicates the process of the addition of a concrete observer class with its corresponding states, i.e., a concrete observer class shown in Figure 7 (B), is inserted into the Observer pattern shown in Figure 7 (A). The resulting Observer pattern is shown in Figure 7 (C). Figure 7 (A) is the UML representation of an Observer pattern with five classes, Subject, ConcreteSubject, Observer, ConcreteObserver1 and ConcreteObserver2. Class ConcreteSubject is a subclass of class Subject, whereas the ConcreteObserver1 and ConcreteObserver2 classes are the subclasses of class Observer. Every concrete class has two states represented as properties in Figure 7 (A), i.e., property s1 and property s2. Figure 7 (B) represents a concrete observer class which is added into the original Observer pattern. This model is automatically generated when the addition process is triggered. From the pattern-level evolution of the Observer pattern, when a concrete observer class is added into the Observer pattern, a generalization relationship between the concrete observer class and class which plays the role of Observer in the Observer pattern should be added. In addition, the states in existing concrete observer classes should also be added. Figure 7 (B) shows a class with the name of ConcreteObserver3 and two attributes s1 and s2 is added into the Observer class. The evolved Observer pattern with three concrete observer classes is generated after the addition process is performed.

The second possible evolution of the Observer pattern is to add a new subject state to be observed. This new state should also be added into all concrete observer classes and concrete subject class to keep the Observer pattern consistent. The designer can use the interface

shown in Figure 5 to perform such evolution. In this case, the *State* on the left panel may be selected so that the designer can input the name of the new state and click on the *Add* button to perform the evolution.

Figure 8 depicts this pattern-level evolution process of the Observer design pattern. The flow labeled with the plus (+) sign indicates the process of the addition of model elements, i.e., all the model elements recorded in Figure 8 (B) will be added into the original Observer pattern shown in Figure 8 (A) and the evolved Observer pattern is depicted in Figure 8 (C). Figure 8 (A) shows the tree representation of the Observer pattern with two states in the concrete subject and the concrete observer classes as shown in Figure 2. To add a state in the concrete subject class of the Observer pattern, a state should be added simultaneously in the concrete observer classes to keep the observer pattern consistent. In this example, state s3 should be added into classes ConcreteSubject, ConcreteObserver1 and ConcreteObserver2. Figure 8 (B) shows the model elements which are needed to be added into the Observer pattern. After performing model transformation, the evolved Observer pattern is depicted in Figure 8 (C) with three states in classes ConcreteSubject, ConcreteObserver1 and ConcreteObserver2.

### 3.2 The Removal of Model Elements

Similar to the addition of a model element, the removal of a model element may also result in the removal of a group of model elements simultaneously. For instance, the Observer pattern has two possible choices of removing a model element, one is removing a concrete observer class from the Observer pattern, and the other is removing a state from a concrete observer class. If we remove a concrete observer class, ConcreteObserver3, from the Observer pattern shown in Figure 3, the attributes of ConcreteObserver3 class and the generalization relationship associated with ConcreteObserver3 class with class Observer should also be removed. The evolved Observer pattern is shown in Figure 2. If we remove state s3 from the ConcreteSubject class in the Observer pattern shown in Figure 4, the state s3 should also be removed from the classes ConcreteObserver1 and ConcreteObserver2, and the application result of the Observer pattern is shown in Figure 2.

Figure 6 shows the user interface to perform the deletion action in the Observer pattern. When the designer selects the *Remove* action under the possible evolutions of the Observer pattern in the left panel of the interface, all removable classes, such as ConcreteObserver1, ConcreteObserver2 and ConcreteObserver3, are displayed in a dropdown list in the right panel. The designer may select one of the classes, e.g. ConcreteObserver3, and click on the *Remove* button. The chosen class, its attributes, and its relations with other classes are removed from the design automatically.

Figure 8 also depict such deletion process. The flow labeled with the minus (-) sign in Figure 8 indicates the process of the deletion of model elements. Figure 8 (C) is the original Observer pattern with three states. Figure 8 (B) shows the model elements which are needed to be deleted from the Observer pattern. After performing the delete action, the resulting Observer pattern is shown in Figure 8 (A).

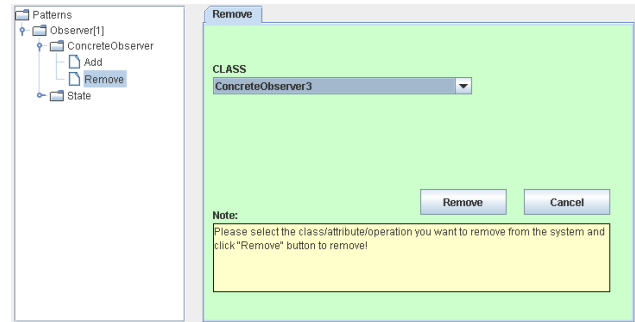


Figure 6 User Interface for Removing Model Element

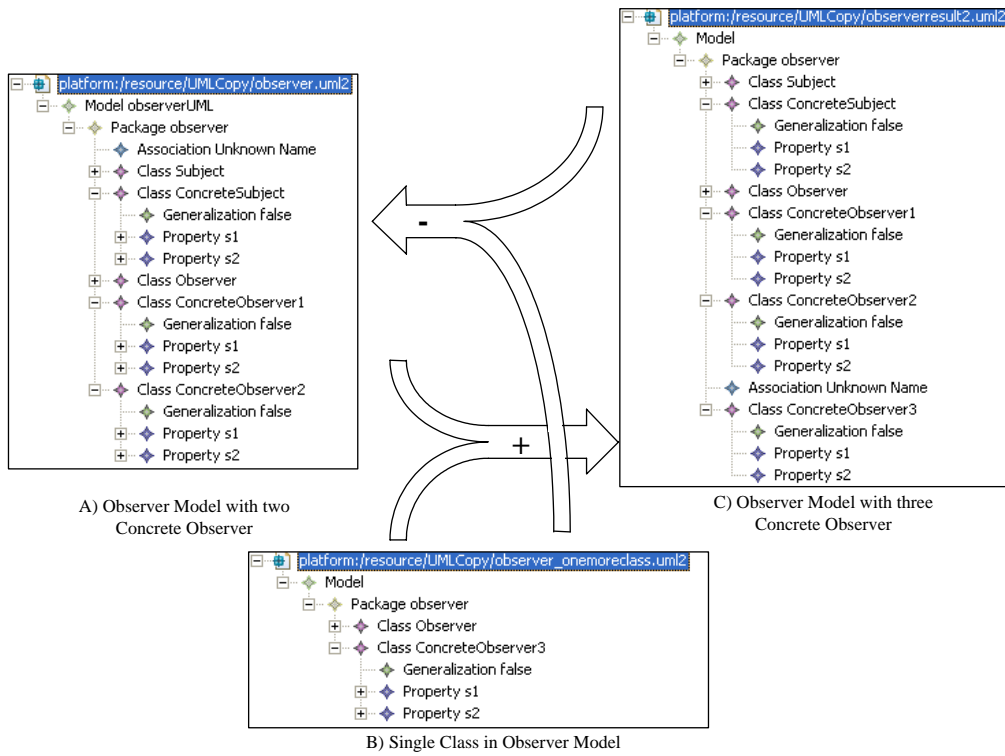
### 3.3 Model Transformation Environment

Figure 9 depicts IBM Eclipse interface for design pattern evolution process. The left panel of interface shows the project packages, including java application source, libraries used in the package, source model, target model and pre-defined transformation rules. The invocation of the application is through an application invocation interface. There exist two ways to invoke the model transformation application. One way is to invoke the application through Mapping Rule Editor, and the other way is through a java application. The former is usually used to develop and debug transformation rules and the latter has a wider usage, such as the java application can be embedded into other applications, such as the user interface shown in Figure 5.

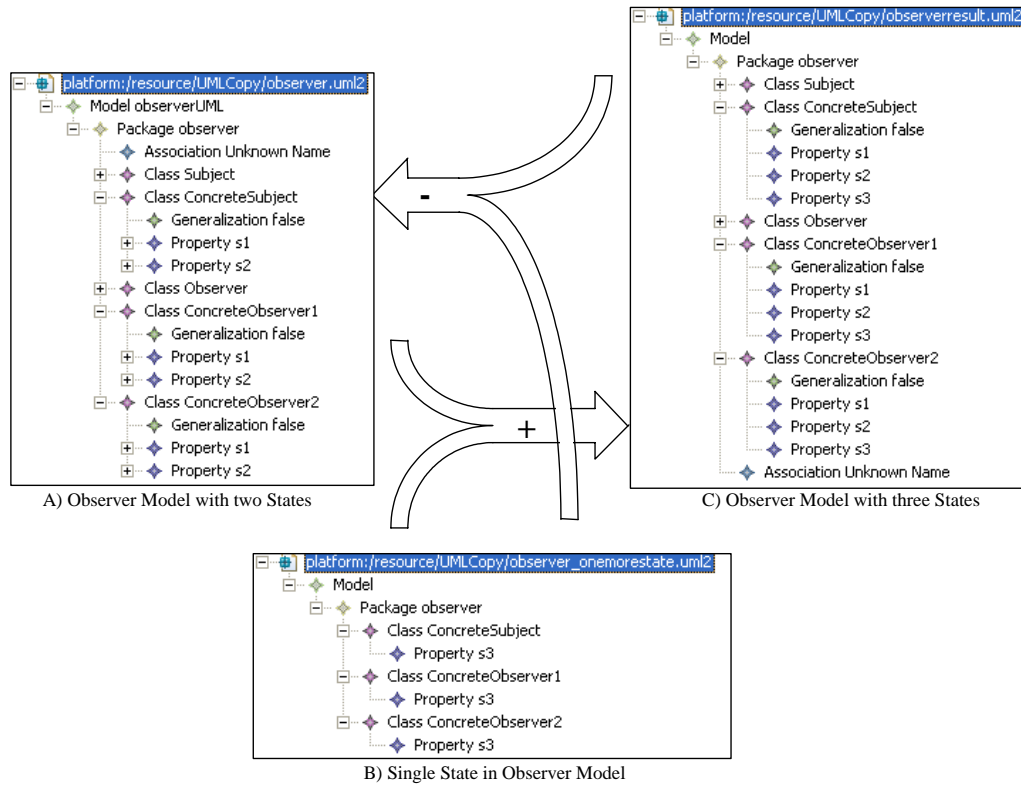
## 4. Related Work

Design pattern evolutions in software development processes are discussed in [10], where software development processes are considered as the evolutions of analysis and design patterns. The evolution rules are specified in Java-like operations to change the structure of patterns. Although some primitive-level evolution rules are introduced, there is no discussion on pattern-level evolution rules. In addition, our approach automates the evolutions based on model transformations by QVT.

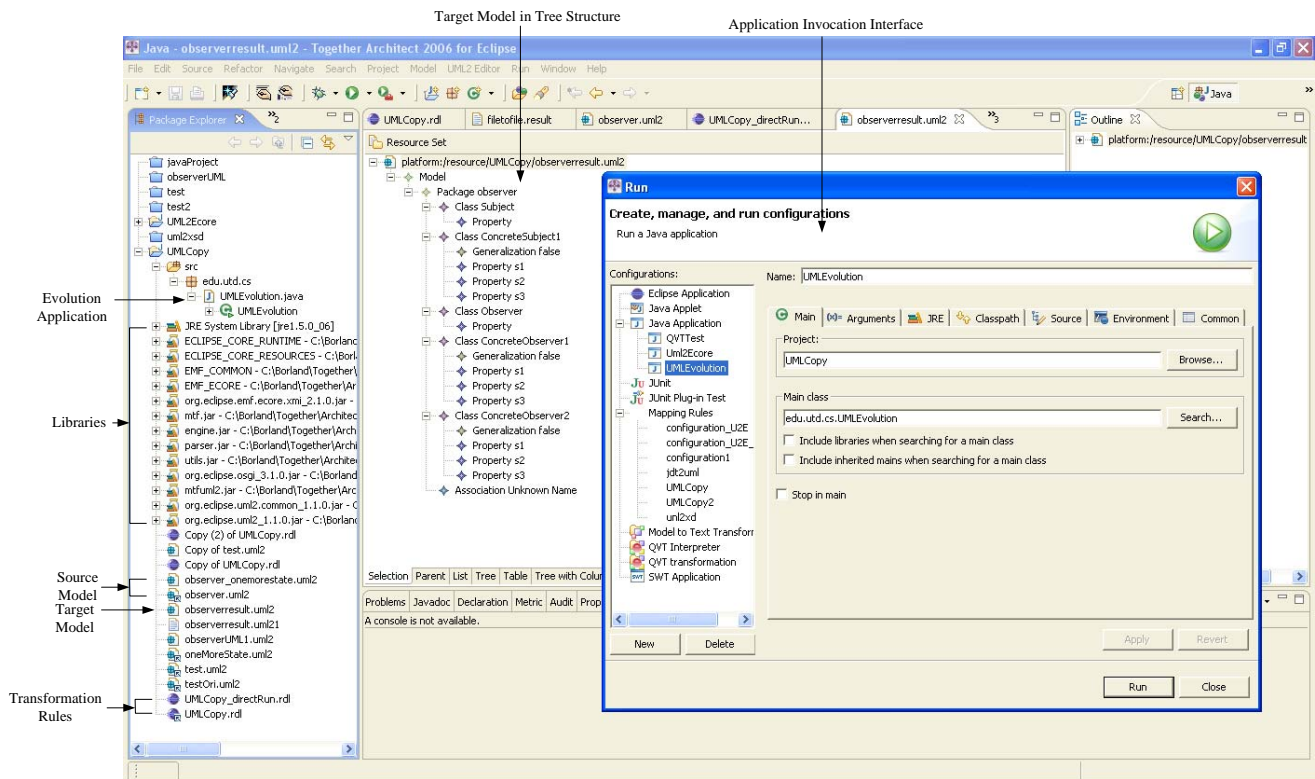
Improving software system quality by applying design patterns in existing systems has been discussed in [2]. When the user selects a design pattern to be applied in a chosen location of a system, automated application is supported by applying transformations corresponding to the mini-patterns. The main goal of their software evolution is to apply design patterns in existing systems, whereas our evolution goal is to change the design patterns that have already been applied in a system.



**Figure 7 The Evolution of Observer Design Pattern (Addition/Removal of a Class)**



**Figure 8 The Evolution of Observer Design Pattern (Addition/Removal of a State)**



**Figure 9 The Design Pattern Evolution Process Interface**

The tool support for UML model evolution is provided in [9], 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.

Mazon et al. [11] applied MDA to data warehouse (DW) development framework to address the design of the whole DW. The framework consists of five layers and each layer represents a part of the whole DW system. Each layer has different viewpoints, including the Platform Independent Model (PIM), Platform Specific Model (PSM) and Computation Independent Model (CIM), according to MDA. Each layer and viewpoints requires different model formalisms. Once PIM models in each layer are developed, other models can be transformed based on relational layer of QVT.

D'Ambrogio [3] applied QVT model transformation to automatically build a performance model from the UML model. The performance model is necessary to effectively validate the performance of a software system through its development lifecycle. Building performance models based on model transformation obtains a high degree of automation in the generation of performance models from software development models.

Kalnins et al. [7][8] proposed a graphical procedural transformation language MOLA. The model transformation defined by MOLA is a sequence of graphical statements linked by arrows. MOLA is more suitable for the transformation between two models, such as transformation from UML diagram to RDBMS schema. Other model transformation languages, such as

QVT-Merge[19], ATL[14], MTF[16], Tefkat[20], and Fujaba Story diagrams (SDM)[15], which are either textual or graphical languages, address the transformation from one model to another. Muller et al. [12] also proposed a model transformation language (Kermeta) to better describe the behavioral aspect of model transformation. In contrast, our purpose is to describe the pattern evolution and automate this process based on QVT.

## 5. Conclusion

Since the evolution information of a design pattern is generally implicit in the descriptions of the pattern, a designer has to dig into the pattern descriptions to understand the particular ways of evolutions encapsulated in 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 find and correct. Third, the evolution processes are not reusable if not documented. As discussed previously, many of the evolution processes recur in different patterns.

In this paper, we define the evolution processes of each design pattern as model transformation rules based on QVT. These transformation rules are defined based on our classifications of two-level transformations: the primitive level and the pattern level. We also provide

tool support for the automation of such model transformation processes. In this way, the evolution information of each design pattern is not only made explicit to the designers, but also automated for them to apply as a transformation.

## References

- [1] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide* (Addison-Wesley, 1999).
- [2] M. Ó Cinnéide and P. Nixon. Automated Software Evolution Towards Design Patterns, *Proceedings of the International Workshop on the Principles of Software Evolution*, pp162-165, Vienna, Austria, September, 2001.
- [3] Andrea D'Ambrogio, A Model Transformation Framework for the Automated Building of Performance Model from UML Models, *Proceeding of the 5<sup>th</sup> International Workshop on Software and Performance*, pp 75-86, Palma, Illes Balears, Spain, July 2005
- [4] J. Dong, S. Yang and K. Zhang, A Model Transformation Approach for Design Pattern Evolutions, *the Proceedings of the Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS)*, pp 80-89, Germany, March 2006.
- [5] J. Dong, S. Yang and D. T. Huynh, Evolving Design Patterns Based on Model Transformation, *Proceedings of the Ninth IASTED International Conference on Software Engineering and Applications (SEA)*, pp 344-350, USA, Nov. 2005.
- [6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995).
- [7] A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA, *Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004)*, pp. 14-28, Linköping, Sweden, June 2004
- [8] A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA: Extended Patterns. *6<sup>th</sup> International Baltic Conference DB@IS 2004*, IOS Press, FAIA vol. 118, 2005, pp. 169-184
- [9] 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*, Maui, Hawaii, Jan. 2001.
- [10] T. Kobayashi and M. Saeki. Software Development Based on Software Pattern Evolution, *Proceedings of the Sixth Asia-Pacific Software Engineering Conference (APSEC)*, pp 18-25, Takamatsu, Japan, 1999.
- [11] Jose-Norberto Mazon, Juan Trujillo, Applying MDA to the Development of Data Warehouses, *Proceedings of the 8<sup>th</sup> ACM International Workshop on Data Warehousing and OLAP*, pp 57-66, Bremen, Germany, November, 2005
- [12] P.-A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, J.M Jezequel, On Executable Meta\_Languages Applied to Model Transformations, *Proceedings of INRIA Workshop of Model Transformations In Practice*, Jamaica, October 2005
- [13] Gustavo Rossi, Daniel Schwabe, and Alejandra Garrido. Design Reuse in Hypermedia Applications Development. *Proceedings of the ACM International Conference on Hypertext*, pages 57–66, April 1997.
- [14] ATL <http://www.sciences.univ-nantes.fr/lina/atl/>
- [15] Fujaba User Documentation <http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf>
- [16] MTF <http://www.alphaworks.ibm.com/tech/mtf>
- [17] Model Driven Architecture. <http://www.omg.org/mda/>
- [18] OMG QVT specification <http://www.omg.org/docs/ptc/05-11-01.pdf>
- [19] QVT-Merge, <http://www.omg.org/docs/ad/05-03-02.pdf>
- [20] Tefkat <http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/>
- [21] W3C, Extensible Markup Language (XML), <http://www.w3.org/>
- [22] W3C, XSL Transformations (XSLT), <http://www.w3.org/>