Design Pattern Evolutions In QVT

Jing Dong [a, 1], Yajing Zhao [a], Yongtao Sun [b]

[a] Computer Science Department, University of Texas at Dallas, Richardson, TX 75083, USA

[b] American Airlines, 4333 Amon Carter Blvd, Fort Worth, TX 76155, USA

Abstract

One of the main goals of design patterns is to design for change. Many design patterns leave some room for future changes and evolutions. The application of design patterns leads to adaptable software since the design pattern instances could be changed with minimal impact on other parts of the system. Such changes, called the evolution of a design pattern instance in this paper, typically involves the addition or removal of a group of model elements, such as classes, attributes, operations, and relationships. However, the possible evolutions of each design pattern are often not explicitly documented. Missing part of the evolution process may result in inconsistent evolution. In this paper, we present our approach to assisting the evolution processes of design patterns by model transformation technology. We provide a formal foundation for our approach by defining the predicates that can be used to describe the properties of each design pattern, software system design, and design pattern evolutions. We also provide tool support to automate the evolution processes based on the Query, View, Transformation (QVT). A case study on a large open-source software system is conducted to illustrate and evaluate our approach.

KEYWORDS

Design pattern evolution; Model Transformation; UML; QVT; Model Driven Architecture

[1] Corresponding email: jdong@ieee.org

1 Introduction

Large software applications become more and more difficult to maintain and change due to their increasing size and complexity. Making changes on software systems requires designers to understand the original software system design and source code. Comparatively, it is less costly to perform changes on the software system at the design level than at the implementation level. To cope with the complexity and improve software quality, design patterns [16][4], which capture expert design experience, have been proposed to be reused in software system designs. Their applications, called design pattern instances, help partitioning and encapsulating software designs into stable part and changeable part. Most of the design pattern instances encapsulate future changes that may only affect a limited part of the entire design. By applying design patterns, the change impact on a software design could be minimized and thus the adaptability of the software design could be improved. Making changes on a design pattern instance is called design pattern evolution process, which is an important part of the maintenance and evolution of a software system at the design level. It is a non-trivial process, which is better to have assistance and validation.

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 should not violate the constraints or properties of design patterns. The evolution process of a design pattern may involve the addition or removal of several correlated elements to ensure the properties of the pattern. Without a deep understanding of design patterns, a designer may perform inconsistent evolutions by missing parts of the evolution process which result in the violations of pattern properties. Thus, it is important to have, in the documentation of the design pattern, the evolution information of the patterns.

In this paper, we propose an approach to defining and automating design pattern evolution process. We define pattern evolution as the evolution of a pattern instance in a software system design. The input of the pattern evolution process is a software system design. After the evolution, a pattern instance in the design is changed. There are typically three kinds of changes for design pattern instance evolutions. The first kind of change is to add one or a group of model elements into the original software system design. Model elements refer to classes, attributes, operations, and relationships. The second kind of action is to remove model element(s) from the original software system design. The third kind of action is to replace existing model elements by new model elements. Replacing model elements could be considered as removing the existing model elements from the software system design and then adding new model elements to the software system

design; thus is not being discussed in this paper. The output of the evolution process is a new software system design with the modified design pattern instances.

To present our approach formally, we define the predicates to describe pattern properties. We also use the predicates to specify software system design before and after the evolution of design pattern instance. The differences between the specifications for the software system design before and after an evolution indicate the changes performed during the evolution process. To realize the evolution process, we define model transformation rules according to the QVT standard. The rules are able to transform a software system design model to a new model by adding or removing model elements. We automate such evolution processes in the Eclipse environment. In addition, we provide a case study on a large open-source software system to illustrate and evaluate our approach.

The remainder of this paper is organized as follows. The next section presents the background knowledge. Section 3 presents the motivation of our approach based on an example. Section 4 introduces the formalism of our pattern evolution approach. Section 5 discusses the implementation of the approach, i.e., the tool support for automating the pattern evolution processes. Section 6 presents a case study on a large open-source software system. The last two sections cover related work and conclusion.

2 Background

The Model Driven Architecture (MDA) [39][34] supports software development based on models as primary artifacts to raise the level of abstraction. Thus, the level of reuse is raised accordingly since high-level software models are as reusable as low-level 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.

Model transformation is one of the key components in the Model Driven Architecture. A model transformation takes as input a model conforming to a given meta-model and produces as output another model conforming to the same or a different meta-model. It defines the process of converting source model into target model. A transformation is also a model that conforms to a given meta-model.

The Query, View, Transformation (QVT) [40] is a specification by the Object Management Group (OMG). It defines a standard way to transform source models into target models by model transformation rules. It standardizes the model transformation by splitting model transformations into two-level architecture: relations meta-model and core meta-model. The relation meta-model specifies, in a user-friendly way, a model transformation as a set of relations which must hold

for a successful model transformation. It supports complex object pattern matching and object template creation. It creates trace classes and trace instances to record the transformation execution. The core meta-model is defined using minimal extension to essential MOF (EMOF) and Object Constraint Language (OCL) [33]. It defines a transformation as a set of mappings and defines trace classes explicitly as MOF models. It also creates and deletes trace instances explicitly. Compared to the relation meta-model, its semantics can be defined more simply. The core model may be implemented directly, or used as a reference for the semantics of relation models. The relations meta-model can be mapped to the core meta-model, using the transformation language itself.

QVT defines three domain-specific languages, namely *Relations*, *Core* and *Operational Mapping*. The *Relations* and *Core* are two declarative languages at two different abstraction levels, the relation meta-model level and the core meta-model level. The *Relations* language has a textual and a graphical concrete syntax. The *Core* language is as powerful as the *Relations* language, though with simpler semantics. The *Operational Mapping* language is an imperative language that extends both *Relations* and *Core*. Its syntax provides constructs commonly found in imperative languages, such as loops and conditions. It is able to define unidirectional transformations using the imperative approach. In addition, it can be used to complement relational transformations by implementing the relations with imperative operations.

IBM's Model Transformation Framework (MTF) is an implementation of the QVT standard. It is an Eclipse based tool that helps designers to develop, execute and debug transformations. MTF transformations can be performed 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. All other models in EMF are mapped to Ecore either explicitly or implicitly. Transformations in MTF are defined in a declarative way. The transformation rules are defined as mapping between the objects in the source model and those in the target model. The MTF transformation engine performs the transformation based on the pre-defined transformation rules 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 the mapping between source and target model objects, some mappings may become inconsistent with respect to the pre-defined transformation rules. The reconciliation stage solves these inconsistencies by generating missing elements, modifying existing elements, and deleting elements.

The Unified Modeling Language (UML) [3] is the *de facto* standard for object-oriented modeling. It can be used to model software system designs as well as to represent design patterns. The evolution of a design pattern instance residing in a software design may be considered as a transformation from the source software design model to a new target software design model, where both the source and the target models conform to the same meta-model, the UML meta-model.

The XML Metadata Interchange (XMI) [43] is an interchange format for metadata in terms of the Meta Object Facility (MOF) [39]. XMI specifies how UML models are mapped into a XMI file. By representing a UML model in the XMI format, the UML model is manageable with the rich collection of XML related techniques and tools.

The pattern evolution approach discussed in this paper uses model transformation techniques. In particular, the model transformation part of this approach conforms to the QVT standard and uses IBM MTF as the execution platform. The data processed in the model transformation is software system model including design pattern instances. The approach uses UML and XMI to represent the data in a standard machine-processable format.

3 Motivation

Design for change is one of the main goals of design patterns. Each design pattern documents the structure and behavior of a group of classes. Most design patterns encapsulate some possible evolutions or changes. These evolutions allow the applications of a design pattern to be changed in some particular ways such that the impact of the change is minimal. However, the evolutions of each design pattern are typically documented implicitly, which makes it difficult to change the software system design. Any mistaken or incomplete change may cause design defects, such as inconsistencies in the design or the violation of the design pattern properties. Therefore, it is important to explicitly document the possible evolutions of each design pattern and automate such evolution processes by defining the possible evolutions.

In [12][13], we have examined the changes and classified the pattern-level evolutions into five categories: the *independent, packaged, class group, correlated classes,* and *correlated operations/attributes*. The *independent* pattern-level evolution describes the addition or removal of one class and the corresponding relationships between this class and other classes in the original pattern instance. This class is independent in the sense that the addition or removal of the class does not cause any other changes on the pattern instance. This kind of evolution appears in several design patterns, for example, in the Mediator and Façade patterns. The *packaged* pattern-level evolution is the addition or removal of one class, the required attributes and/or operations of this class, and the corresponding relationships between this class and

other classes in the original pattern instance. This kind of evolutions is different from the first kind in the sense that the addition of an empty class may violate the pattern properties. The required attributes and operations must be added so as to maintain the pattern properties. The *class group* 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. The *correlated classes* pattern-level evolution is the addition or removal of a group of correlated classes, the attributes and operations of the classes, as well as the corresponding relationships between this group of classes and other classes. Some pattern, such as the Abstract Factory pattern, requires the correspondence relations between classes. If one class is added or removed, the classes that have one-to-one correspondence relationship with that class must also be added or removed; otherwise, it causes inconsistency. The addition or removal of this group of classes should not affect the internal attributes and operations of other classes in the original design pattern applications. The *correlated attributes/operations* pattern-level evolution is the addition or removal of a group of classes, as well as the addition or removal of some attributes or operations in the classes of the original pattern instance. Some design patterns, such as the Adapter, Visitor and Abstract Factory patterns [16], require the one-to-one correspondence between one class and the attribute/operation of another class. Therefore, the addition or removal of a class requires the addition or removal of the attribute/operation that has the corresponding relationship with it. This kind of evolution may change the internal attributes and operations of the classes of the original pattern instance, which is different from the *correlated classes* pattern evolution.

We described these pattern-level evolutions based on some primitive-level evolutions that describe the basic transformations performed during the evolution process of a design pattern. These basic transformations include the addition or removal of a model element, such as class, operation, attribute, association, generalization, aggregation, composition, realization, and dependency. These basic transformations, which become the building blocks of the pattern-level evolution, are defined as model transformation rules as discussed in Section 5.3.

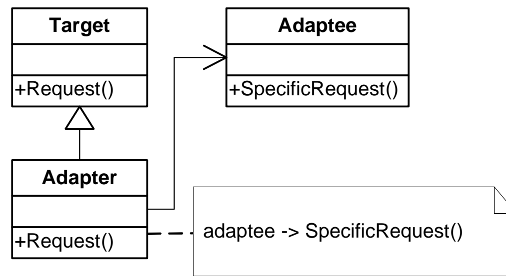


Figure 1 The Adapter Pattern

Consider the Adapter pattern [16] whose class diagram is shown in Figure 1. The Adapter pattern has its own minimal set of properties, to which all its instances should conform. An instance is no longer a valid Adapter pattern, if it violates any Adapter pattern property. The detail definition of the minimal set of properties is given in Section 4.2.

Figure 2 shows an instance of the Adapter pattern extracted from the design of Java.awt package [37]. The ArcIterator class that plays the role of Adapter wraps the transform() operation in the AffineTransform class that plays the role of Adaptee to fit the interface of the Target interface, the currentSegment() operation in the PathIterator class.

In this example, one possible evolution of the Adapter pattern is to add a new Adaptee (e.g., GeneralPath) that includes one operation: getWindingRule(), as shown in Figure 3. To maintain the conformance to the Adapter pattern properties, a new Adapter class (GeneralPathIterator) with one operation (getWindingRule()) needs to be added, and an operation (getWindingRule()) needs to be added to the existing Target class (PathIterator). This pattern evolution involves the addition of a group of classes along with their attributes and operations, which fits into the *correlated operations/attributes* evolution category.

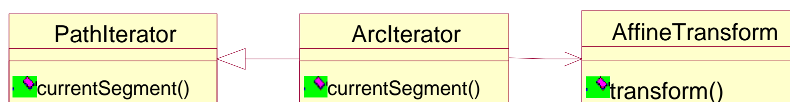


Figure 2 An Application of The Adapter Pattern

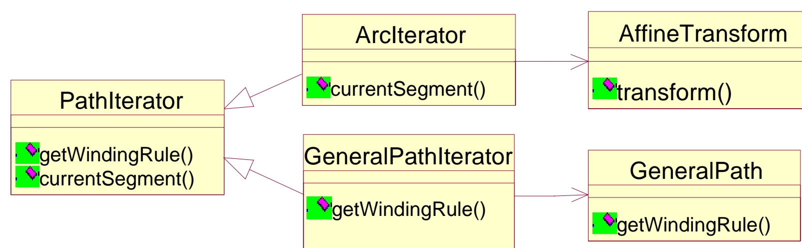


Figure 3 Adapter Pattern Evolution

On the other hand, the Adapter pattern has two possible choices for removing a model element: removing an Adapter class or removing an Adaptee class from the Adapter pattern. Suppose we remove an Adaptee class, GeneralPath, from the Adapter pattern instance shown in Figure 3. To maintain the Adapter pattern properties, the GeneralPathIterator class with its operation, the corresponding operation of the GeneralPath class, the corresponding operation of the PathIterator class, and the generalization relationship between the GeneralPathIterator and PathIterator classes should also be removed. The evolved Adapter pattern is shown in Figure 2. It includes two instances[2] of the Adapter pattern.

Without correct understanding of the Adapter pattern, a designer may end up with adding or removing only the Adaptee class, instead of the entire group of model elements. Manual pattern evolution may cause inconsistency or violation of design pattern properties. Therefore, there is a need to document the possible evolutions for each design pattern, provide a tool to assist the evolution process, and use the tool to set restrictions on the possible evolutions that can be performed by a designer.

4 Specifying Design Pattern Properties and Evolutions

To present the design pattern evolution process formally, we introduce some predicates in Section 4.1. The predicates could be used to define properties of design patterns, as discussed in Section 4.2. They could be used to formally specify the software system designs, and thus to describe the design pattern evolutions, as discussed in Section 4.3.

4.1 Predicates for Design Pattern Properties

We define three groups of predicates: predicates for defining model elements, predicates for defining relationships, and negation predicates.

4.1.1 Predicates for Defining Model Elements

Predicates in this section define whether a design pattern or a piece of software system design has a specific class, whether a class has a specific operation, whether an operation has a particular parameter, the properties of the classes, attributes, and operations.

Definition 1. HAS Predicate: a predicate that defines the existence of model elements.

For example, the following predicates are HAS predicates.

[2] Although it can be considered as two variations of one instance of the Adapter pattern as well, we consider it as two instances of the Adapter pattern for consistency.

Predicate 1. hasClass (d, c): Design d includes Class c.

Predicate 2. hasOperation (c, o): Class c has operation o.

Predicate 3. hasOperationSet (c, os) ::= $\forall o_i \in os \bullet$ hasOperation (c, o_i): each operation o_i in os is a method of Class c.

Predicate 4. hasAttribute (c, a): Class c has attribute a.

Predicate 5. hasStatement (o, s): Operation o has a statement s.

Predicate 6. hasReturnValue (o, Ob): Operation o has Ob as return value.

Predicate 7. hasReturnType (o, T): Operation o has T as return type.

Predicate 8. hasParameterList (o, l): l is the parameter list of Operation o.

Predicate 9. hasParameterType (l, t, k): t is the type of the kth parameter in the parameter list l.

Predicate 10. hasName (e, n): Pattern/Model element (class, operation or attribute) e has n as its name.

Definition 2. IS Predicate: a predicate that defines properties of the model elements.

For example, the following predicates are IS predicates.

Predicate 11. isAbstract (x): Class/Operation x is abstract.

Predicate 12. isConcrete (x): Class/Operation x is concrete.

Predicate 13. isPublic (x): Attribute/Operation x is public.

Predicate 14. isPrivate (x): Attribute/Operation x is private.

Predicate 15. isProtected (x): Attribute/Operation x is protected.

Predicate 16. isStatic(x): Attribute/Operation x is static.

Predicate 17. isType (ob, t): Object ob is of type t.

4.1.2 Predicates for Defining Relationships

Predicates in this category define the constraints for the relationships between classes, relationships between operations and classes, relationships between operations.

Predicates 18 through 27 are the class level predicates, which define relationships between classes.

Predicate 18. generalize (c_1, c_2): Class c_1 is a subclass of class c_2.

Predicate 19. groupGeneralize (cs, c) ::= $\forall c_i \in cs \bullet$ generalize (c_i, c): each class c_i in cs is a subclass of c.

Predicate 20. associate (c_1, c_2): Class c_1 keeps a reference to class c_2.

Predicate 21. aggregate (c_1, c_2): Class c_1 maintains a reference(s) to c_2, and class c_2 is a part of c_1 semantically.

Predicate 22. oneToMore (c_1, c_2): The multiplicity of association or aggregation relationship from class c_1 to class c_2 is 1:m.

Predicate 23. oneToOne (c_1, c_2): The multiplicity of association or aggregation relationship from class c_1 to class c_2 is 1:1.

Predicate 24. moreToMore (c_1, c_2): The multiplicity of association or aggregation relationship from class c_1 to class c_2 is m:m.

Predicate 25. moreToOne (c_1, c_2): The multiplicity of association or aggregation relationship from class c_1 to class c_2 is m:1.

Predicate 26. create (c_1, c_2): Class c_1 is responsible for creating Class c_2.

Predicate 27. correspondingRelated (p (set_1, set_2)) ::= $\forall e_1 \in set_1$, $\exists e_2 \in set_2 \bullet p$ (e_1, e_2): This predicate is used specially in the Abstract Factory pattern where there shall be at least one create method for each family of products.

Some of the relationships can be detailed at the operation level, such as the create relationship and the delegation relationship.

Predicate 28. create (c_1, o, c_2): Operation o in class c_1 creates class c_2.

Predicate 29. delegate (c_1, o_1, c_2, o_2): Operation o_1 in class c_1 forwards request to operation o_2 in class c_2.

4.1.3 Negation Predicates

Some design patterns require certain relations not to exist between certain classes, attributes, or operations. In the Adapter pattern, for example, there is a generalization relationship from Adapter to Target. Hence, there shall not be a generalization relationship from Target to Adapter, to avoid a circular generalization relationship. Negation relationships are used to set this kind of restrictions.

Predicate 30. noDirectAccess (c_1, c_2): Class c_1 does not have direct access to class c_2, the attributes or the operations of c_2.

Predicate 31. differentInterface (c_1, o_1, c_2, o_2) = (hasParameterList(o_1, l) $\wedge \neg$ hasParameterList(o_2, l)) $\vee$ (hasReturnType(o_1, t) $\wedge$ $\neg$ hasReturnType(o_2, t)): Operation o_1 in class c_1 has different interface (parameter list and return type) from operation o_2 in class c_2, i.e., l is the parameter list of operation o_1, but not that of operation o_2; or t is the return type of operation o_1, but not that of operation o_2.

Predicate 32. noGeneralize (c_1, c_2) = $\neg$ (generalize (c_1, c_2)): There shall not be a generalization relationship from class c_1 to c_2.

Predicate 33. noAggregate (c_1, c_2) = $\neg$ (aggregate (c_1, c_2)): There shall not be an aggregation relationship from class c_1 to c_2.

4.2 Specifying Design Pattern Properties

The predicates defined in the previous section are general enough to specify all GoF patterns. In this section, we use the predicates to specify the properties of the Adapter pattern, as an example. The Adapter pattern, as shown in Figure 1, has three roles, Target, Adapter, and Adaptee. The Target is an abstract or a concrete class, the Adapter is a concrete class, and the Adaptee is a concrete class. The Target and Adapter must have at least one common operation that call an operation defined in the Adaptee. The Adaptee is not a subclass of the Target. The Adapter provides an interface different from that provided by the Adaptee or the Client does not have direct access to the Adaptee. We define these properties as follows:

$\exists$ p, c_1, c_2, c_3, o_1, o_2, o_3 $\bullet$ (hasName (p, “Adapter Pattern”) $\wedge$

hasClass (p, c_1) $\wedge$ hasName (c_1, “Target”) $\wedge$ (isAbstract (c_1) $\vee$ isConcrete(c_1)) $\wedge$

hasClass (p, c_2) $\wedge$ hasName (c_2, “Adapter”) $\wedge$ isConcrete (c_2) $\wedge$

hasClass (p, c_3) $\wedge$ hasName (c_3, “Adaptee”) $\wedge$ isConcrete (c_3) $\wedge$

hasClass (p, c_4) $\wedge$ hasName (c_4, “Client”) $\wedge$

hasOperation (c_1, o_1) $\wedge$ hasName (o_1, “Request”) $\wedge$ hasReturnType (o_1, t) $\wedge$ hasParameterList (o_1, l) $\wedge$ isAbstract(o_1) $\wedge$

hasOperation (c_2, o_2) $\wedge$ hasName (o_2, “Request”) $\wedge$ hasReturnType (o_2, t) $\wedge$ hasParameterList (o_2, l) $\wedge$ isConcrete(o_2) $\wedge$

hasOperation (c_3, o_3) $\wedge$ hasName (o_3, “SpecificRequest”) $\wedge$ isConcrete (o_3) $\wedge$

generalize (c_2, c_1) $\wedge$ associate (c_2, c_3) $\wedge$ delegate (c_2, o_2, c_3, o_3) $\wedge$ noGeneralize (c_3, c_1) $\wedge$

(differentInterface (c_2, o_2, c_3, o_3) $\vee$ noDirectAccess (c_4, c_3)))

4.3 Specifying Design Pattern Evolution

Pattern evolution defines the changes of a design pattern instance in a software system design by adding or removing model elements. The input of the pattern evolution process is a software system design. The output of the evolution process is a new software system design with the modified design pattern instance. We use the predicates to describe software system design. The design in Figure 2, for example, could be described as follows:

$\exists$ $c_1, c_2, c_3, o_1, o_2, o_3$ ● (hasClass (*system*, c_1) $\wedge$ hasName (c_1, "PathIterator") $\wedge$ isConcrete (c_1) $\wedge$

hasClass (*system*, c_2) $\wedge$ hasName (c_2, "ArcIterator") $\wedge$ isConcrete (c_2) $\wedge$

hasClass (*system*, c_3) $\wedge$ hasName (c_3, "AffineTransform") $\wedge$ isConcrete (c_3) $\wedge$

hasOperation (c_1, o_1) $\wedge$ hasName (o_1, "currentSegment") $\wedge$ isAbstract (o_1) $\wedge$

hasOperation (c_2, o_2) $\wedge$ hasName (o_2, "currentSegment") $\wedge$ isConcrete (o_2) $\wedge$

hasOperation (c_3, o_3) $\wedge$ hasName (o_3, "transform") $\wedge$ isConcrete (o_3) $\wedge$

generalize (c_2, c_1) $\wedge$ associate (c_2, c_3) $\wedge$ delegate (c_2, o_2, c_3, o_3))

Design pattern evolution involves the addition or removal of model elements. We can also represent the elements to be added or removed by the predicates. In this way, the possible evolutions for each pattern can be formally specified. For each design pattern, we identify the possible evolutions, define them using the predicates, and represent them in XML file. The possible evolutions in XML format, which can be read and processed automatically, will be used in the concrete implementation of the pattern evolution approach. Each GoF pattern may evolve in several possible ways in terms of the five types of pattern evolutions we defined previously. We presented a list of all possible evolutions of each GoF pattern [12][13]. For example, the Adapter pattern has several possible evolutions. One of the evolutions is to add a new Adaptee together with a new Adapter, which belongs to the *correlated attributes/operations* pattern-level evolution as discussed in Section 3. This can be specified by the predicates as:

$\exists$ $c_1, c_2, c_3, o_1, o_2, o_3$ ● (hasClass (*system*, c_1) $\wedge$ hasName (c_1, c_1-name) $\wedge$ (isAbstract(c_1) $\vee$ isConcrete (c_1)) $\wedge$

hasClass (*system*, c_2) $\wedge$ hasName (c_2, c_2-name) $\wedge$ isConcrete (c_2) $\wedge$

hasClass (*system*, c_3) $\wedge$ hasName (c_3, c_3-name) $\wedge$ isConcrete (c_3) $\wedge$

hasOperation (c_1, o_1) $\wedge$ hasName (o_1, o_1-name) $\wedge$ isAbstract (o_1) $\wedge$

hasOperation (c_2, o_2) $\wedge$ hasName (o_2, o_2-name) $\wedge$ isConcrete (o_2) $\wedge$

hasOperation (c_3, o_3) $\wedge$ hasName (o_3, o_3-name) $\wedge$ isConcrete (o_3) $\wedge$

generalize (c_2, c_1) $\wedge$ associate (c_2, c_3) $\wedge$ delegate (c_2, o_2, c_3, o_3))

where c_i-name (i = 1,2,3) and o_j-name (j = 1,2,3) are class names and operation names, respectively, which are actually provided by a designer during the evolution process.

The possible pattern evolutions are pre-defined to allow a designer to add or remove a group of model elements consistently. They prevent a designer from missing parts of the evolution. For the example in Section 3, when a new Adapter (e.g., GeneralPathIterator) is added, a designer has to provide the evolution information according to the pre-defined evolution. The predicates representing the model elements to be added could be obtained by providing the class names (c_i-name) and operation names (o_j-name) in the pre-defined set of predicates as follows:

$\exists$ c_1, c_2, c_3, o_1, o_2, o_3 • (hasClass (*system*, c_1) $\wedge$ hasName (c_1, "PathIterator") $\wedge$ isConcrete (c_1) $\wedge$

hasClass (*system*, c_2) $\wedge$ hasName (c_2, "GeneralPathIterator") $\wedge$ isConcrete (c_2) $\wedge$

hasClass (*system*, c_3) $\wedge$ hasName (c_3, "GeneralPath") $\wedge$ isConcrete (c_3) $\wedge$

hasOperation (c_1, o_1) $\wedge$ hasName (o_1, "getWindingRule") $\wedge$ isAbstract (o_1) $\wedge$

hasOperation (c_2, o_2) $\wedge$ hasName (o_2, "getWindingRule") $\wedge$ isConcrete (o_2) $\wedge$

hasOperation (c_3, o_3) $\wedge$ hasName (o_3, "getWindingRule") $\wedge$ isConcrete (o_3) $\wedge$

generalize (c_2, c_1) $\wedge$ associate (c_2, c_3) $\wedge$ delegate (c_2, o_2, c_3, o_3))

Pattern evolution could be represented by the addition or removal of the predicates into or from the predicate set of the original design. Therefore, the addition of model elements could be achieved by the conjunction of the predicate set of the original design with that of the model elements. The removal of model element could be accomplished by the subtraction of the predicate set of the model elements from that of the original design. For example, the software system design after evolution including the original design and the model elements to be added is shown in Figure 3, which could be described by the conjunction of the predicates representing the original design and the predicates representing the additional model elements as follows:

$\exists$ c_1, c_2, c_3, c_4, c_5, o_1, o_2, o_3, o_4, o_5, o_6 • (hasClass (*system*, c_1) $\wedge$ hasName (c_1, "PathIterator") $\wedge$ isConcrete (c_1) $\wedge$

hasClass (*system*, c_2) $\wedge$ hasName (c_2, "ArcIterator") $\wedge$ isConcrete (c_2) $\wedge$

hasClass (*system*, c_3) ∧ hasName (c_3, "AffineTransform") ∧ isConcrete (c_3) ∧

hasClass (*system*, c_4) ∧ hasName (c_4, "GeneralPathIterator") ∧ isConcrete (c_4) ∧

hasClass (*system*, c_5) ∧ hasName (c_5, "GeneralPath") ∧ isConcrete (c_5) ∧

hasOperation (c_1, o_1) ∧ hasName (o_1, "currentSegment") ∧ isAbstract (o_1) ∧

hasOperation (c_2, o_2) ∧ hasName (o_2, "currentSegment") ∧ isConcrete (o_2) ∧

hasOperation (c_3, o_3) ∧ hasName (o_3, "transform") ∧ isConcrete (o_3) ∧

hasOperation (c_1, o_4) ∧ hasName (o_4, "getWindingRule") ∧ isAbstract (o_4) ∧

hasOperation (c_4, o_5) ∧ hasName (o_5, "getWindingRule") ∧ isConcrete (o_5) ∧

hasOperation (c_5, o_6) ∧ hasName (o_6, "getWindingRule") ∧ isConcrete (o_6) ∧

generalize (c_2, c_1) ∧ associate (c_2, c_3) ∧ delegate (c_2, o_2, c_3, o_3) ∧

generalize (c_4, c_1) ∧ associate (c_4, c_5) ∧ delegate (c_4, o_5, c_5, o_6))

Describing software system designs by predicates allows us to view the evolution as adding or removing predicates.

5 Pattern Evolution in QVT

In this section, we introduce our pattern evolution approach based on model transformation in detail. We first provide an overview of our approach. We then discuss the Pattern Evolver User Interface, the transformation rules, and the transformation environment. Finally, we use an example to illustrate the evolution process execution results.

5.1 Overview of the Approach

We have automated these pattern-level evolution processes based on the Extensible Stylesheet Language Transformation (XSLT) in [11][12]. In an XSLT [44] transformation, each primitive-level evolution has an associated XSLT transformation rule and is expressed with the tag <XMI.add> or <XMI.delete> for the addition or deletion of a model element, respectively. The pattern-level evolutions are represented in XMI similarly by grouping the XMI specifications of the corresponding primitive-level evolutions. For each pattern-level evolution, there is a group of associated XSLT transformation rules that can add/remove the corresponding model elements of this pattern-level evolution into/from the original UML model in XMI format. The pattern evolution is performed one model element at a time. For each primitive-level evolution, e.g., adding/deleting an attribute/operation to/from a class, the XSLT

transformation rule is applied by scanning the entire software system design to locate the class. If two or more attributes/operations need to be added into a class, scanning the entire software system design needs to be done multiple times, which wastes a considerable amount of execution time, hence impacting the performance of pattern evolution. This could be a dramatic drawback because the software system design is usually very large. In this paper, we investigate the QVT-based model transformations for the pattern-level evolutions based on our initial study in [14]. Unlike using procedure language to specify the transformation rules in the XSLT-based pattern evolution approach, the QVT-based approach specifies transformation rules using declarative rule language.

Figure 4 depicts the overall architecture of our approach for automated evolutions of design patterns based on model transformations. Design pattern applications in a software design are normally modeled using the UML. Since UML diagrams are typically persisted in proprietary formats of the corresponding UML tools, a standard XMI format has been defined for the UML diagrams and the plug-in of these UML tools has been developed to export UML diagrams into XMI format. Because the IBM Eclipse Modeler can translate software system UML design diagram to textual XMI format based on the UML 2.0 definitions, the *Original System Design* is translated into the *Source Model (in XMI-UML2 format)* by IBM Modeler in Figure 4.

We provide the *Pattern Evolver User Interface,* an interactive interface to perform software system design evolution. It takes the *Source Model (in XMI-UML2 format)* and a definition of the possible pattern evolutions in XML as inputs. It generates a list of design pattern instances applied in the original design model and displays them. According to the possible pattern evolution definition, it also generates a list of possible evolutions for each design pattern instance. Designers can select a design pattern instance from the list and a type of pattern-level evolutions they want to perform. They provide necessary information for the evolution, *i.e.* the names of the elements to be changed. To add a class, designers should enter the name of the class. To add attributes or operations to the class, they also have to provide the names of the attributes/operations. To delete a model element, they need to provide the name and other related information. Based on the input, the interface generates the *Segment Model (in XMI-UML2 format)*, which contains the elements to be changed (to be added or to be deleted). The *Source Model* can be formally specified in the predicates defined in the previous section. For example, the properties of the Adapter pattern have been specified in Section 4.2. An instance of the Adapter pattern can be the *Source Model* that is specified in the predicates defined in Section 4.2.

Similarly, the *Segment Model* can be specified in the predicates defined in Section 4. For instance, a *Segment model* of the Adapter pattern is specified in the predicates defined in Section 4.3.

The *Source Model (in XMI-UML2 format)* and the *Segment Model (in XMI-UML2 format)* are implicitly converted into the *Source Model (Ecore)* and the *Segment Model (Ecore)*, respectively, which are Ecore models defined in the IBM EMF. These two models are taken as input by the *Transformation Engine* in IBM EMF to produce the *Target Model (in XMI-UML2 format)* in UML 2.0 format. The *Model Transformation Rules* are applied during the transformation.

The Model Transformation Rules specify how the model elements are changed (added or removed) from the original software system design. It specifies how the *Source Model* and the *Segment Model* are combined or how the elements in the *Segment Model* are subtracted from the *Source Model*.

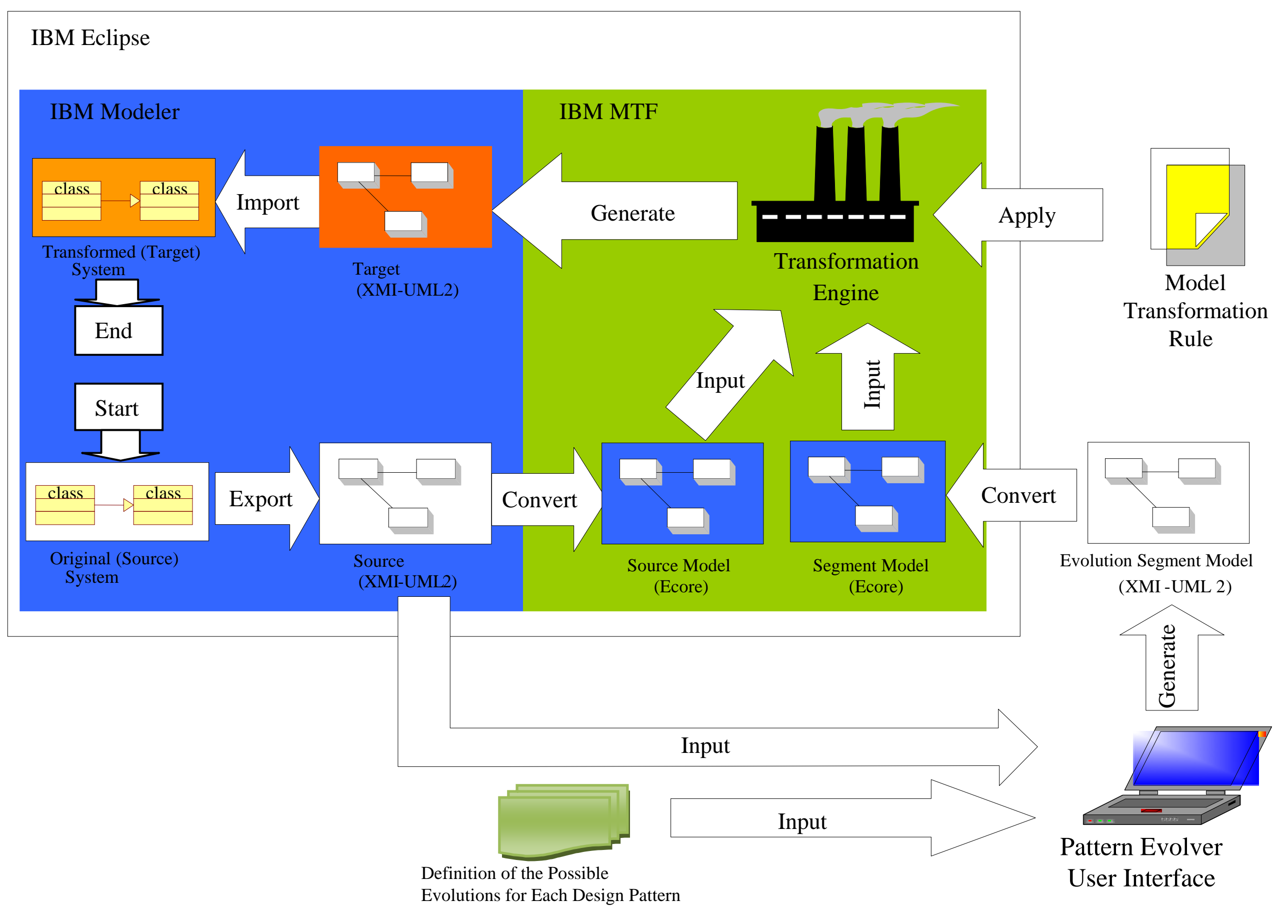


Figure 4 the Overall Architecture of the Approach

In the following sections, we discuss our *Pattern Evolver User Interface*, the QVT based model transformation rules, and how to use the Transformation Engine to perform the evolution.

5.2 Pattern Evolver User Interface

The *Pattern Evolver User Interface* takes as input a software system design expressed in the XMI-UML2 format and displays design pattern instances in the user interface. It allows designers to view all the instances. In addition, it takes as input a definition of the possible evolutions for each design pattern, according to which it generates a list of evolution choices. In this way, it educates designers about the possible pattern evolutions and allows them to choose and perform a pattern evolution in a batch. Once designers have provided the evolution information, it records and outputs the information in the XMI-UML2 format.

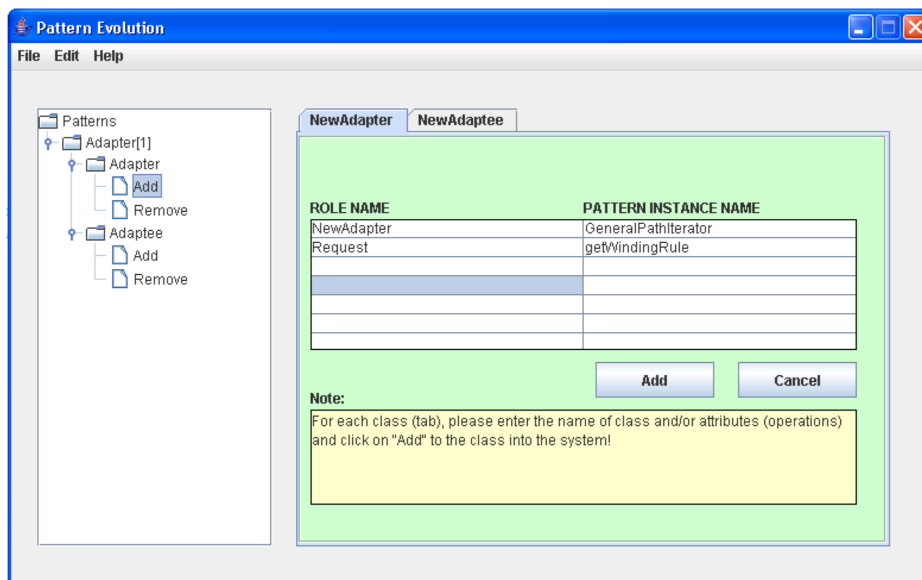


Figure 5 User Interface for Adding Model Element

The *Pattern Evolver User Interface* provides two kinds of interfaces for the changes, one for addition and the other for removal. The replacement could be performed by removing and then adding. Figure 5 illustrates the interface for addition. The left panel of the interface shows there is one Adapter pattern instance in the input software system design. The items under each design pattern are possible pattern-level evolutions for that design pattern. For the Adapter pattern, e.g., there are two possible evolutions, adding/removing an Adapter class and adding/removing an Adaptee class. Each possible evolution has two actions, *Add* and *Remove*. When a designer selects the *Add* action in the Adapter pattern, the right panel will display a table asking designers for the necessary information to perform the *Add* action, with one class on each tab. The designer enters the name of Adapter class and the name of the operation that plays a role of Request in the Adapter pattern on the *NewAdapter* tab and the names of Adaptee class and corresponding operation that plays a role of

SpecificRequest in the Adapter pattern on the *NewAdaptee* tab. After all information has been entered, the designer can click on the *Add* button. When the *Add* button is clicked, the evolution information is recorded as an evolution segment model in the XMI-UML2 format. Since the pre-defined evolutions define the possible evolutions that can be performed, they limit the actual segment models that could be defined. Later on, both the evolution segment model and the original software system model are sent to the transformation engine that performs the process according to the addition rules. We will introduce the transformation rules in the next section. When a designer selects the *Remove* choice under the possible evolutions of the Adapter pattern in the left panel of the interface, the removal interface will be displayed, as shown in Figure 6. In the removal interface, all removable classes, such as GeneralPathIterator, GeneralPath, ArcIterator and AffineTransform, is displayed in a dropdown list in the right panel. A designer may select one of the classes, e.g. GeneralPathIterator, and click on the *Remove* button. The chosen class, together with its attributes and operations, and its relations with other classes, is recorded as a model in the XMI-UML2 format, a segment model. If the removal of the chosen class requires the deletion of other related class in order to maintain the pattern property, the information of the related classes is also recorded in the segment model.

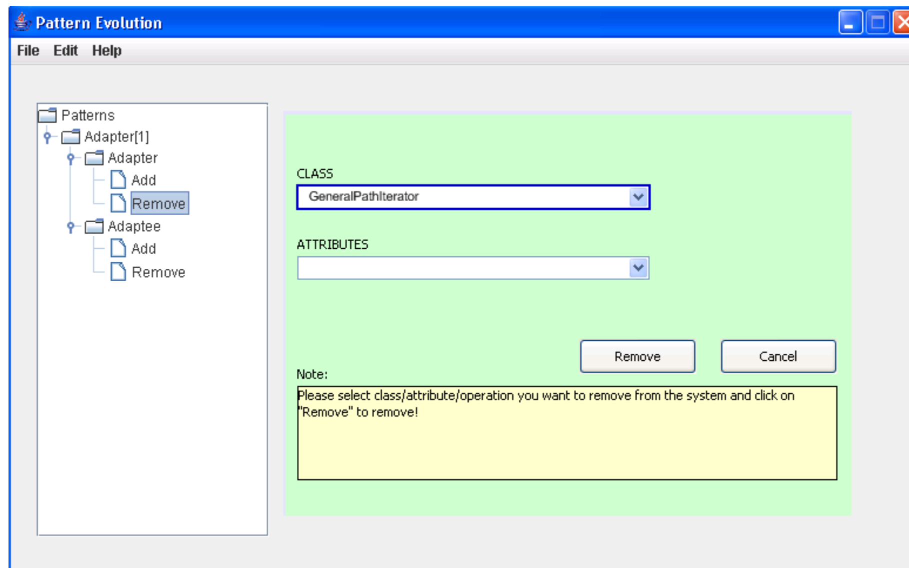


Figure 6 User Interface for Removing Model Element

5.3 QVT Based Model Transformation Rules

We define model transformation rules in QVT, which are applied in two types of changes: adding or removing model elements. The inputs of the transformation include the source model that represents the original design and the segment

model that represents the model elements to be added into or removed from the source model. The output of the transformation is the target model. All three models conform to the same UML meta-model. The model transformation rules are defined in the MTF as a set of relations.

```
8 relate FileToFile (uml:IFile umlsrc, uml:IFile umlSeg, uml:IFile umltgt)
9 {
10    ModelToModel(umlsrc.model, umlSeg.model, umltgt.model)
11 }
12 relate ModelToModel (uml:Model srcModel, uml:Model segModel, uml:Model tgtModel)
13   when equals(srcModel.name, tgtModel.name) | equals (segModel.name, tgtModel.name)
14 {
15    PkgToPkg(over srcModel.ownedMember, over segModel.ownedMember, over tgtModel.ownedMember)
16 }
17 relate PkgToPkg (uml:Package srcPackage, uml:Package segPackage, uml:Package tgtPackage)
18   when equals (srcPackage.name, tgtPackage.name) | equals (segPackage.name, tgtPackage.name)
19 {
20   PkgToPkg (over srcPackage.ownedMember, over segPackage.ownedMember, over tgtPackage.ownedMember),
21   ClassifierToClassifier (over srcPackage.ownedMember, over tgtPackage.ownedMember),
22   ClassifierToClassifier (over segPackage.ownedMember, over tgtPackage.ownedMember),
23   AssociationToAssociation (over srcPackage.ownedMember, over tgtPackage.ownedMember),
24   AssociationToAssociation (over segPackage.ownedMember, over tgtPackage.ownedMember),
   …
28 }
29 abstract relate ClassifierToClassifier(uml:Classifier srcClassifier, uml:Classifier tgtClassifier)
30   when equals (srcClassifier.name, tgtClassifier.name)
31 relate ClassToClass extends ClassifierToClassifier (uml:Class srcClass, uml:Class tgtClass)
32 {
33   ClassifierToClassifier (over srcClass.nestedClassifier, over tgtClass.nestedClassifier),
34   ordered GeneralizationToGeneralization [0..1] (over srcClass.generalization, over tgtClass.generalization),
35   ordered AssociationToAssociation (over srcClass.Association, over tgtClass.Association),
36   ordered DependencyToDependency (over srcClass.Dependency, over tgtClass. Dependency),
37   PropertyToProperty (over srcClass.ownedAttribute, over tgtClass.ownedAttribute),
38   OperationToOperation (over srcClass.ownedOperation, over tgtClass.ownedOperation)
39 }
40 relate InterfaceToInterface extends ClassifierToClassifier (uml:Interface srcInt, uml:Interface tgtInt)
41 {
   …
48 }
49 relate PropertyToProperty(uml:Property srcProp, uml:Property tgtProp)
50   when equals(srcProp.name, tgtProp.name)
…
```

Figure 7 Transformation Rules for Addition

Figure 7 illustrates the transformation rules that we provide for the first type of changes that add model elements into the source model. The *FileToFile* relation (defined by lines 8 through 11) is the starting point, where three files are taken into the transformation procedure. The first two files, *umlSource* and *umlSeg*, are two inputs, one taking the original design model information and the other taking the segment model information. The third file, *umlTarget*, is used as an

output, which is empty before the execution. Line 10 indicates that the *ModelToModel* relation (defined by lines 12 through 16) executes on the models defined in the three files. The *ModelToModel* relation invokes the *PkgToPkg* relation (defined by lines 17 through 28), which defines how the elements in the two inputs are transformed to the output. Line 20 indicates that if the input packages contain sub-package, the elements within the sub-packages are mapped recursively by the very same relation. Lines 21 and 22 indicate that the classifiers (i.e. the classes or the interfaces) in both input models are copied into the target model. If a class is defined in both the source class and the segment class, only one copy is made in the target model due to the following reasons: first, when line 21 executes the MTF engine cannot find a match in the target and thus creates a copy; second, when line 22 executes, the MTF engine matches the newly created copy with the one defined in the segment model and thus does not create another copy. Therefore, the MTF mapping engine has the ability to filter out the duplicate elements. Lines 23 and 24 indicate that the associations in the input models are mapped into the target model.

The *ClassifierToClassifier* relation (defined by lines 29 and 30) has two arguments, the source classifier (*sourceClassifier*) and the target classifier (*targetClassifier*), which represents the source class/interface and the target class/interface of the transformation, respectively. The relation is defined as *abstract* and is extended by two concrete relations, the *ClassToClass* relation and the *InterfaceToInterface* relation. When the *ClassifierToClassifier* relation gets invoked, the MTF engine checks the type of the arguments passed in. If the type is "uml:Class", the mapping relation *ClassToClass* (defined by lines 31 through 39) is performed, and if it is "uml:Interface", the mapping relation *InterfaceToInterface* (defined by lines 40 through 48) is performed.

The relation *ClassToClass* deals with the mapping between classes from the source model and those from the target model. In the body, the relation hierarchically maps the subclasses (line 33), the generalization relationships (line 34), the association relationships (line 35), the dependency relationships (line 36), the attributes (line 37), and the operations (line 38) of the source class to those of the target classes. In line 33, the relation *ClassifierToClassifier* is invoked recursively to map the subclasses. The keyword *over* indicates that the corresponding argument represents collections. A mapping should be created between every element of the collections. The invocation of relation *GeneralizationToGeneralization* in line 34 maps the generalization relationship of the source classes to that of the target classes. The keyword *ordered* allows the generalization relation participants in the target model to maintain the same order as that in the source model. The multiplicity [0..1] in the generalization mapping relation restricts the number of generalization relationships exist between

two classes to be at most one. The association and dependency mappings are defined similarly in lines 35 and 36. In addition, the relation *PropertyToProperty* in line 37 and the relation *OperationToOperation* in line 38 map all attributes and operations, respectively, between the classes from the source model and the classes from the target model.

Similarly, the relation *InterfaceToInterface* (lines 40 through 48) 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, association, and dependency relations, attributes, and operations of the source interface to those of the target interface.

The relation *PropertyToProperty* (lines 49 and 50) define the mapping between the class attributes. Other mappings, such as *OperationToOperation*, *GeneralizationToGeneralization*, *AssociationToAssociation*, and *DependencyToDependency*, which are not shown in the figure, can be defined similarly.

```
11  relate FileToFile (ws:IFile umlsrc, ws:IFile umlseg, ws:IFile umltgt) {
12      ModelToModel(umlsrc.model, umlseg.model, umltgt.model)}
13  relate ModelToModel (uml:Model srcModel, uml:Model segModel, uml:Model tgtModel)
14      when equals(srcModel.name, segModel.name) & equals(srcModel.name, tgtModel.name) {
15      PkgToPkg(over srcModel.ownedMember, over segModel.ownedMember, over tgtModel.ownedMember)}
16  relate PkgToPkg (uml:Package srcPackage, uml:Package segPackage, uml:Package tgtPackage)
17      when equals(srcPackage.name, segPackage.name) & equals (srcPackage.name, tgtPackage.name) {
18      PkgToPkg(over srcPackage.ownedMember, over segPackage.ownedMember, over tgtPackage.ownedMember),
19      ClassifierToClassifier(over srcPackage.ownedMember, over segPackage.ownedMember, over tgtPackage.ownedMember),
20      AssociationToAssociation(over srcPackage.ownedMember, over segPackage.ownedMember, over tgtPackage.ownedMember)}
21  abstract relate ClassifierToClassifier(uml:Classifier srcClassifier, uml:Classifier segClassifier, uml:Classifier tgtClassifier)
22     when !equals(segClassifier.name, srcClassifier.name) & equals (tgtClassifier.name, srcClassifier.name)
23  relate ClassToClass extends ClassifierToClassifier (uml:Class srcClass, uml:Class segClass, uml:Class tgtClass) {
24      ClassifierToClassifier(over srcClass.nestedClassifier, over segClass.nestedClassifier, over tgtClass.nestedClassifier),
25      OperationToOperation(over srcClass.ownedOperation, over segClass.ownedOperation, over tgtClass.ownedOperation),
26      PropertyToProperty(over srcClass.ownedAttribute, over segClass.ownedAttribute, over tgtClass.ownedAttribute)}
27  relate InterfaceToInterface extends ClassifierToClassifier (uml:Interface srcInterface, uml:Interface segInterface, uml:Interface
tgtInterface) { ... }
45  relate PropertyToProperty(uml:Property srcProp, uml:Property segProp, uml:Property tgtProp)
46      when (!equals(srcProp.name, segProp.name))&(equals(srcProp.name, tgtProp.name))
47  relate OperationToOperation (uml:Operation srcOperation, uml:Operation segOperation, uml:Operation tgtOperation)
48      when (!equals (srcOperation.name, segOperation.name))& equals(srcOperation.name, tgtOperation.name)
...
```

Figure 8 Transformation Rules for Removal

Figure 8 illustrates the transformation rules that we defined for the type of changes that remove model elements from the source model. The removal rules compare the elements in the source model and those in the segment model, and then map into the target model the elements that appear in the source model but not in the segment model. For example, lines 22, 46 and 48 indicate such comparison and mapping.

We use the IBM Eclipse as our pattern evolution environment. There exist two ways to invoke the model transformation application. One way is to invoke the application through the Mapping Rule Editor. 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. For example, the Java application can be embedded into other applications, such as the user interface shown in Figure 5. Figure 9 depicts the IBM Eclipse interface for the design pattern evolution performed through Java application invocation. The left panel of the interface shows the project packages, including Java application source, libraries used in the package, source model, evolution segment model, target model and pre-defined transformation rules. These transformation rules have been discussed previously.

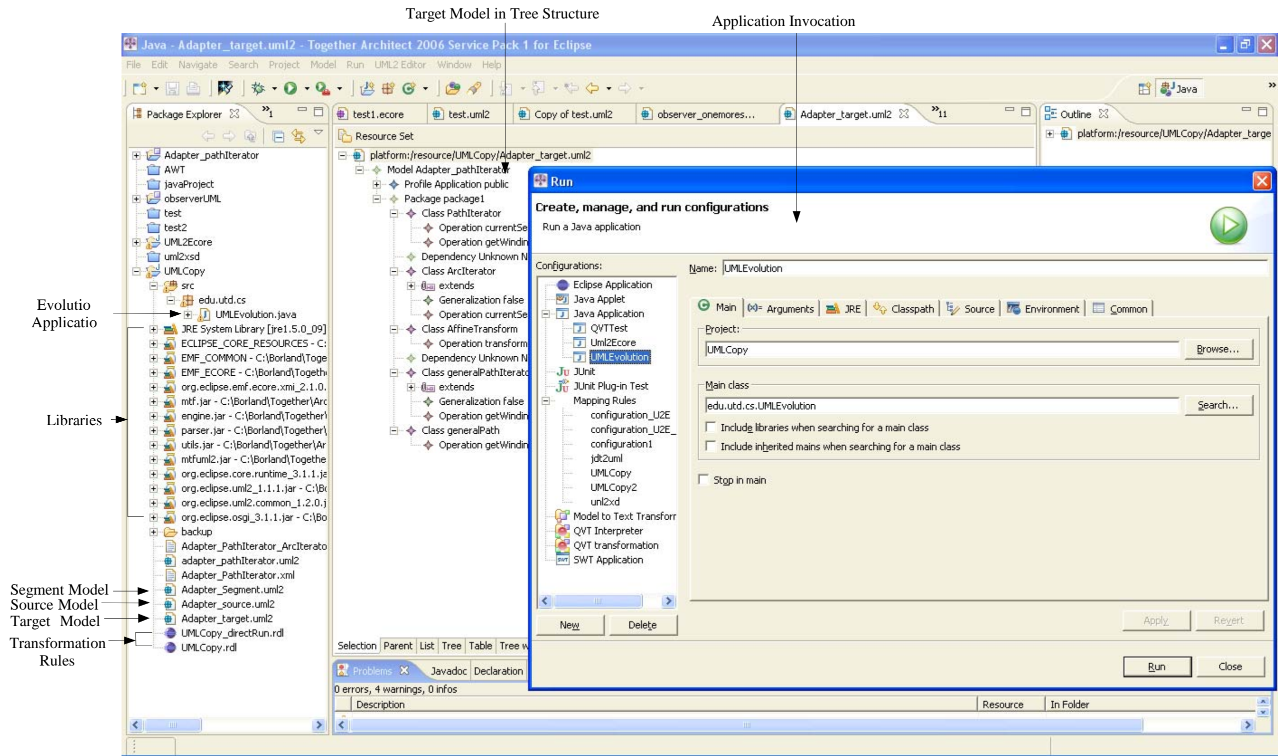
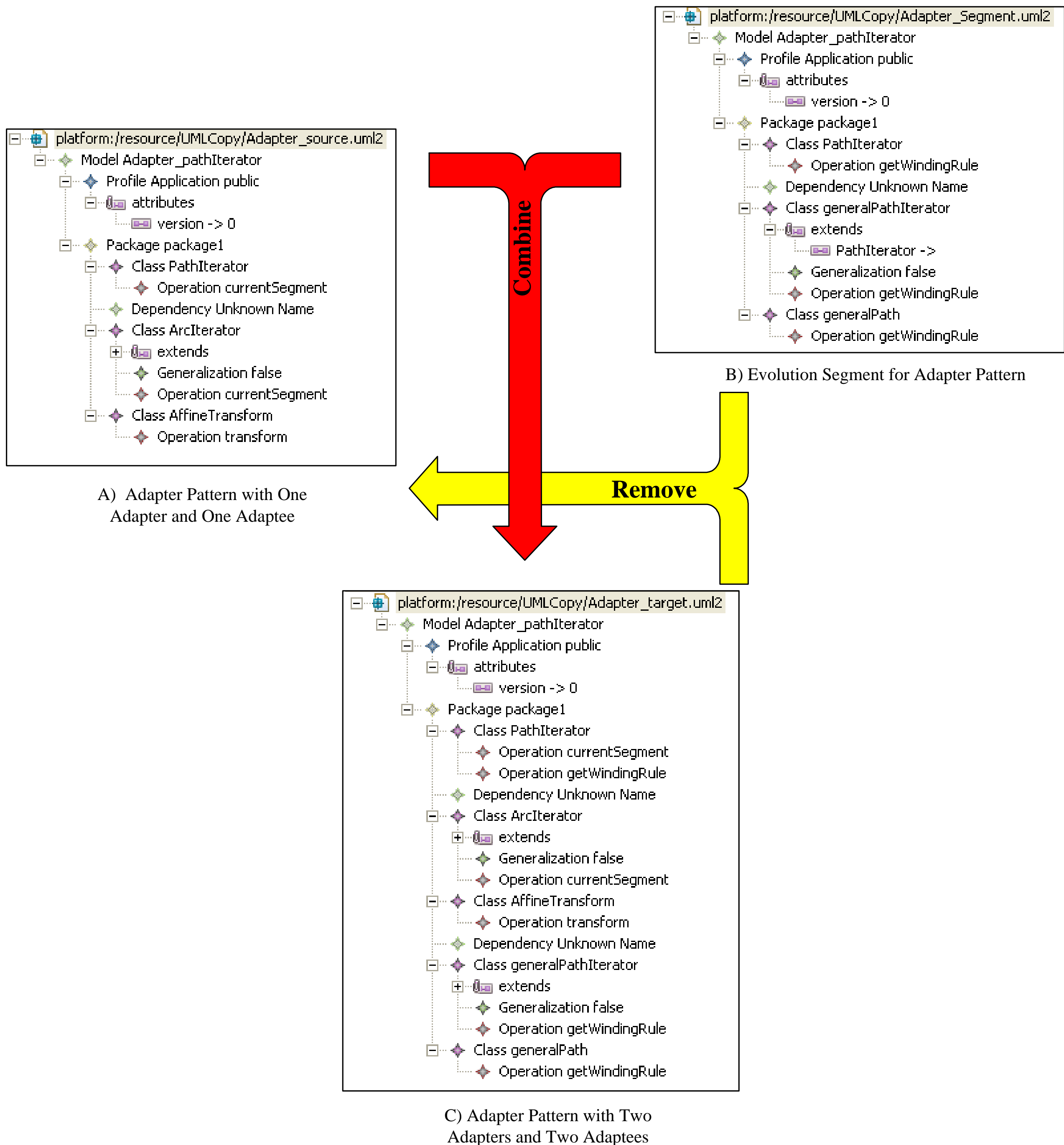


Figure 9 the Design Pattern Evolution Process Interface

The target model is initially an empty model. The type of evolution that adds the segment model into the source model applies the transformation rules defined in the previous section to copy both the elements in the source model (the original software system design) and those in the segment model into the target model. The duplicate elements, which are defined in both the source model and the segment model, are combined. A removal evolution checks every element for their existence in the segment model. If an element in the source model does not exist in the segment model, it copies it into the target model. If it exists in the segment model, it does not copy it. Thus the target model becomes the relative complement of the segment model in the source model.



A) Adapter Pattern with One Adapter and One Adaptee

B) Evolution Segment for Adapter Pattern

C) Adapter Pattern with Two Adapters and Two Adaptees

Figure 10 The Evolution of Adapter Design Pattern

5.4 An Example of Pattern Evolution in QVT

Consider the example software system designs as shown in Figure 2 and Figure 3. Figure 2 describes a software system design with an Adapter pattern instance. The instance has three classes, *PathIterator*, *ArcIterator*, and *AffineTransform*. Class *ArcIterator* is a subclass of class *PathIterator*. Both *PathIterator* and *ArcIterator* classes have an operation named *currentSegment*. The *AffineTransform* class has an operation named *transform*. The software system design can be modeled in the XMI-UML2 format, as shown in Figure 10 (A). Figure 3 describes a software system design with an Adapter pattern instance. The instance has two Adapter classes, *ArcIterator* and *GeneralPathIterator*, and two Adaptee classes, *AffineTransform* and *GeneralPath*, which can be modeled in the XMI-UML2 format, as shown in Figure 10 (C). The evolution segment to be added to Figure 2 or be removed from Figure 3 includes an Adapter class called *GeneralPathIterator* and an Adaptee classes called *GeneralPath*. The information of the evolution segment is provided by a designer and recorded by the *Pattern Evolver User Interface* in the XMI-UML2 format. As illustrated in Figure 10 (B), the segment model has three classes, *PathIterator*, *GeneralPathIterator* and *GeneralPath*. All three classes have operations named *getWindingRule*. The *GeneralPathIterator* class is a subclass of the *PathIterator* class.

The flow labeled with "Combine" in Figure 10 indicates the process of the evolution that adds model elements. During an addition, the transformation takes the source software system model shown in Figure 10 (A) and an evolution segment model shown in Figure 10 (B) as inputs. The transformation engine applies the addition rules defined in Figure 7 as the transformation rules. It combines the model elements in the two input models and generates the result as shown in Figure 10 (C). The flow labeled with "Remove" in Figure 10 indicates the process of the deletion of model elements. In this case, the software system model that is shown in Figure 10 (C) and the evolution segment model that is shown in Figure 10 (B) are taken as inputs. The transformation engine applies the transformation rules to remove the elements in the segment model from the source software system model and generates the result in the target model. The target model will contain an Adapter pattern instance with one Adapter class and one Adaptee class, which is the same as the model in Figure 10 (A).

6 Case Study

In this section, we present a case study on a large open-source software system, the Java.awt package in JDK 1.4 [37]. More specifically, we study the pattern evolutions of this software system using our model transformation techniques and

tools. The goal of this case study is to demonstrate that our pattern evolution approach works fine with real-life software systems.

AWT stands for Abstract Windowing Toolkit, a powerful and flexible Java class library for the development of Graphical User Interface (GUI). It contains rich user components, such as Frames and Panels, which ease GUI development and is able to run on any platform. The definition of each such component also contains a range of actions which can be performed on a specific event associated with the component. The AWT package also contains different layouts for placing components, such as Scrollbar and Button, onto containers like Frame or Panel. These layouts do not depend on window size or screen resolution [37]. The Java.awt package consists of 346 files. These files contain a total of 485 classes (including inner classes) and 111 interfaces. It is large open source software. We used a computer with 3.4 GHz Pentium processor, 1 GB of RAM and Windows XP operating system in our study on the Java.awt package.

Figure 11 A Screenshot of the Design of Java.AWT

Since the Java.awt package is an open-source software system, there is no comprehensive design document. Although many design patterns have been applied by the developers of this software system, pattern-related information is not documented anywhere. One of the methods to recover pattern-related information is to use existing design pattern discovery tools. In particular, we have developed a UML-based design pattern discovery tool, called DP-Miner [9][15]. Using DP-Miner, we recovered 21 Adapter pattern instances, as well as a number of other pattern instances, in the Java.awt package. Two of these Adapter pattern instances, as shown in the previous sections, are composed of five classes, PathIterator as Target, GeneralPathIterator and ArcIterator as Adapter, GeneralPath and AffineTransform as Adaptee. Figure 11 shows the class diagram of the design of the Java.awt package with the aforementioned classes highlighted by a dashed oval. More specifically, the GeneralPathIterator and ArcIterator classes are subclasses of the PathIterator class. The GeneralPathIterator and ArcIterator classes associate with the GeneralPath and AffineTransform classes, respectively. The getWindingRule() operations in the PathIterator and GeneralPathIterator class play the role of Request() whereas the getWindingRule() operation in the GeneralPath class plays the role of SpecificRequest() in the first Adapter instance in the upper part of the oval area in Figure 11. The currentSegment() operations in the PathIterator and ArcIterator class play the role of Request() whereas the transform() operation in the AffineTransform class plays the role of SpecificRequest() in the second Adapter instance in the lower part of the oval area.

In the previous sections, we have shown an example of the addition of a new Adapter pattern instance to a small software system with one Adapter pattern instance. In this section, we illustrate the addition of a new Adapter instance in the design of the Java.awt package. As discussed previously, one possible evolution of the Adapter pattern is a *correlated operations/attributes* evolution that is the addition or removal of a group of classes along with their attributes and operations. For example, the developers may add a new Adapter class that includes one operation, myRequest(). However, the Adapter class cannot be added alone. The corresponding Target and Adaptee classes have to be added or identified. In this case, the PathIterator class is identified as the Target class. Thus there is no need to add a new one. However, the myRequest() operation needs to be added in the existing Target class (PathIterator). In addition, since no Adaptee class is identified, a new Adaptee class is required to be added. Any missing changes of these model elements may result in errors and inconsistencies in the design that are difficult to find and correct.

In this case study, we apply our approach and tool to add a new Adapter instance in the Java.awt package. More specifically, our approach and tool for QVT-based pattern evolution include the following processes.

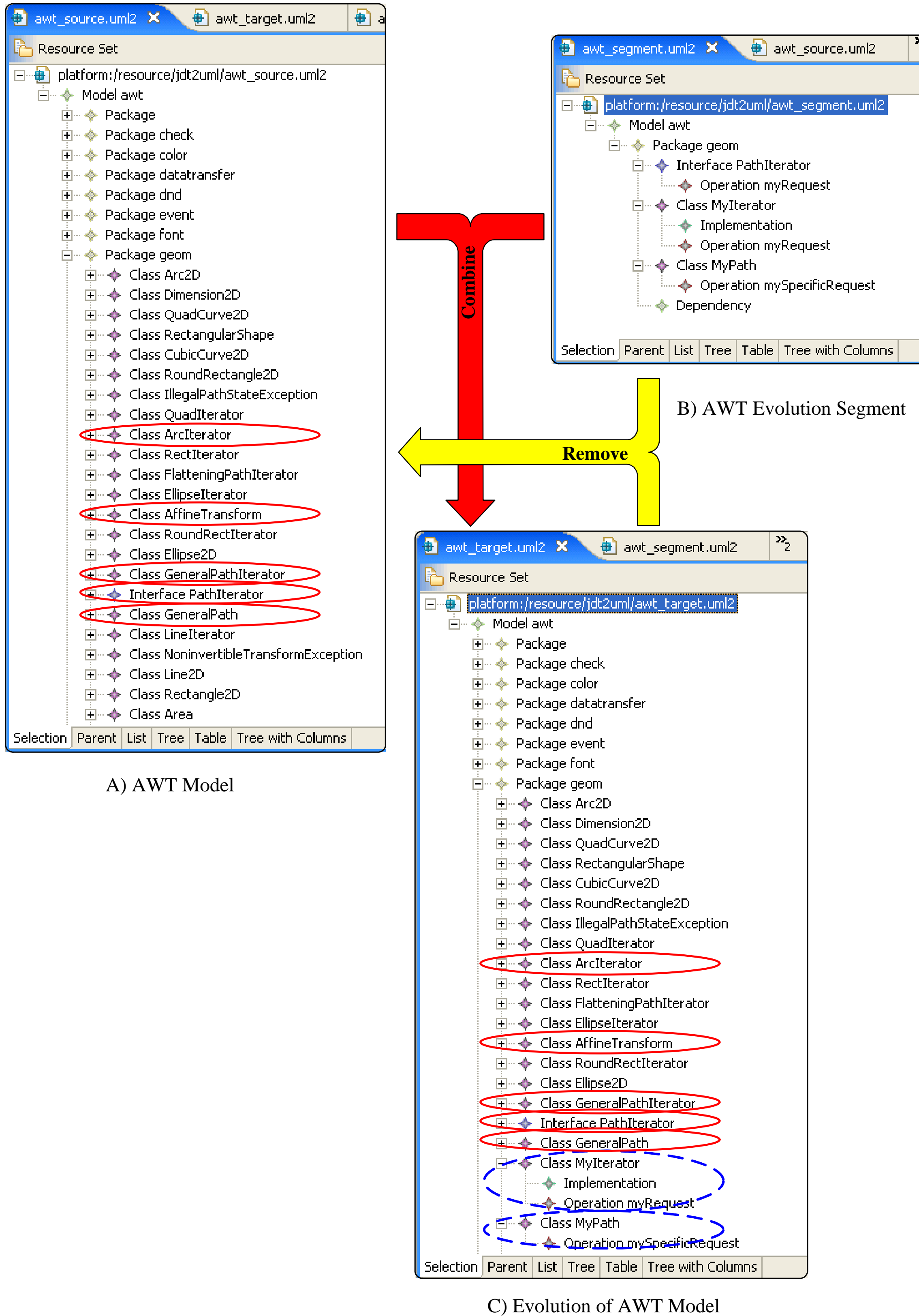


Figure 12 the Evolution Process of Java AWT System

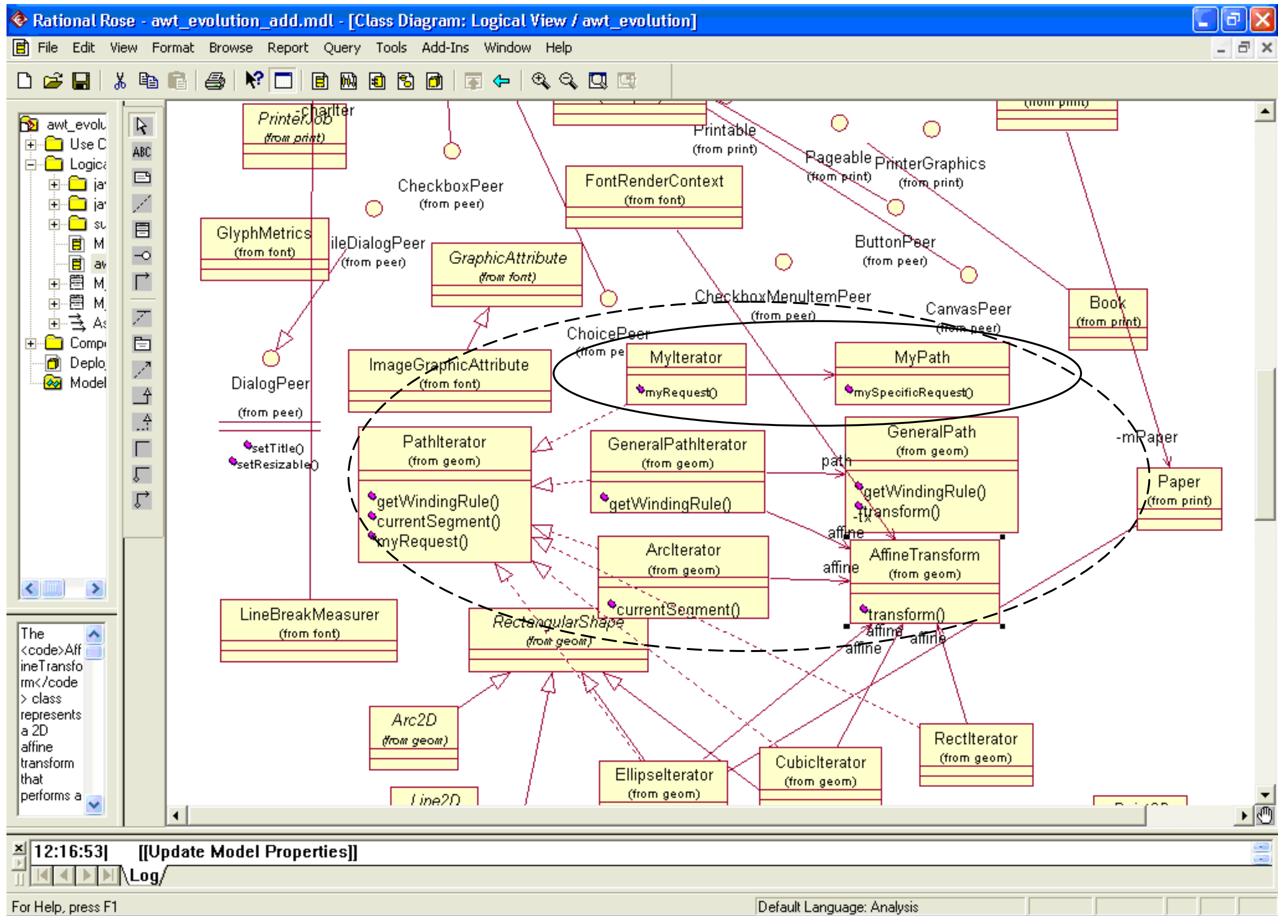


Figure 13 An Evolution of the Design of Java.awt

First, all pattern instances applied in a software system design are identified [9][15] and explicitly displayed. Pattern related information may not be always available in many software systems, especially large real-life software systems. Many approaches [2][9][15][17][29][31] have been proposed to recover such information from software system source code. These recovered pattern instances from a software system are the basis of our approach presented in this paper. In particular, we use our own tool, DP-Miner, to recover the pattern instances in a software system for the following reasons. DP-Miner is also an XMI-based tool that outputs the results of pattern recovery in an XMI file. All pattern-related information in such XMI file can be represented using the stereotypes and tagged values defined in [10]. This XMI file with pattern-related information can be the input of our QVT-based model transformation tool described in this paper, which naturally integrates with our pattern recovery tool. Note that other pattern recovery tools can also be used as the basis of our pattern evolution tool in this paper although some adjustment and adaptation of the results from these tools may be required, which is out of the scope of this paper.

Second, our QVT-based model transformation tool takes the software system design model XMI file with pattern-related information as an input and provides a user interface that allows designers to supply the information of the pattern evolution. The tool displays all the pattern instances available in the AWT package, including 21 instances of the Adapter pattern. With each pattern instance, the tool displays the evolution possibilities. Clicking on an evolution that involves addition action brings up a table, similar to the one shown in Figure 5, where the names of new model elements, which play certain roles in the pattern instance, can be specified by designers. In this case study, a designer adds a new Adapter class to the Adapter instance highlighted in Figure 11. The new Adapter class is named *MyIterator* and the new Request operation is named *myRequest*. Since the Adapter class cannot be added into the design alone, there has to be an Adaptee class to be adapted. The tool, as illustrated in Figure 5, provides another tab (*NewAdaptee*) to request from the designer the information for the new Adaptee. The designer is prompted to provide the names for the new Adaptee class and the new SpecificRequest operation, which, in this case, are *MyPath* and *mySpecificRequest()*, respectively. Our tool requires the designer to provide all information related to the new model elements so that there will be no missing elements. On the other hand, if the designer selects an evolution possibility that removes elements from a pattern instance, all possible pattern participants that could be removed will then be displayed in a dropdown menu, similar to the one shown in Figure 6. Selecting any element will result in the removal of all related elements of the pattern instance.

Third, our transformation rules for adding model elements described in Section 5.3 will be applied after designers provide the information about the new model elements with the guidance of our tool. Figure 12 shows the UML2 models of our model transformation processes in Eclipse. More specifically, Figure 12 A) displays the geom. package of Java.awt that includes the two Adapter pattern instances we are interested, one includes the *PathIterator*, *ArcIterator*, *AffineTransform* classes, the other consists of the *PathIterator*, *GeneralPathIterator*, *GeneralPath* classes, which are highlighted in solid ovals in the diagram. While Figure 12 A) presents the original UML2 model of Java.awt, Figure 12 B) depicts the new model elements of a new Adapter pattern instance that the designer wants to add in Java.awt. The names of these model elements that play certain roles in the new Adapter pattern instance have been provided by the designer in the previous step. In this case, two new classes include *MyIterator* and *MyPath*, each of which has its own operation, *myRequest* and *mySpecificRequest*, respectively. A new operation, *myRequest*, also needs to be added into the *PathIterator* class. In addition, the dependency relationship between classes *MyIterator* and *MyPath* and the generalization relationship between classes *MyIterator* and *PathIterator* need to be added as shown in the diagram. Our

model transformation rules for adding model elements are then applied to combine the two UML2 models shown in Figure 12 A) and B) into a new model shown in Figure 12 C), where the dashed ovals highlight the new addition of the Adapter pattern instance. Figure 13 shows the class diagram of the evolved software system design, with the involved classes highlighted by a dashed oval and the newly added classes highlighted by a solid oval. Removing model elements from the original design model could be achieved similarly by applying the transformation rules for removing model elements, which is responsible for removal. For example, an Adapter instance described in Figure 12 B) can be removed from the model in Figure 12 C), resulting in the model in Figure 12 A).

7 Related Work

The evolution processes of design patterns have been studied in [1], where Prolog [7] is used to capture the structural evolution processes of design patterns. Thus, the evolution of a design pattern application could 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 describe the evolution process as model transformations based on QVT.

Design pattern evolutions in software development processes are also discussed in [24], where software development processes are considered as the evolution analysis of 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 in QVT.

In order to automatically apply design patterns in software system design models, Wang *et al.* [32] propose a design pattern oriented model transformation approach. The transformation framework includes a set of atom mappings and pattern mappings. They combine the two mappings and create the model transformation code template, which can generate XSLT based model transformation code. The template library is easy to be extended for new design patterns. They use the XMI-Light, a middle storage format for many UML tools, as the input and output format. They use XSLT as the transformation technique. Comparatively, our work focuses on the evolution of design patterns instead of their applications. Our evolution approach is based on QVT transformation standard rather than XSLT.

Judson *et al.* propose a design pattern based model transformation and refactoring approach which defines transformation at the meta-model level [18]. They aim at providing a declarable model transformation language to transform software system design based on pre-defined transformation patterns. They specialize the UML meta-model to

characterize source and target models. The transformation patterns can be used to constrain how transformations are defined at the model level and can act as the points against which transformations are checked for conformance. Specifying the transformations patterns as meta-models promotes the controlled evolution of models. The approach has been applied to the Abstract Factory and Visitor patterns. Abstract Factory transformation patterns have been developed for UML class and interaction design models. We investigate the possible evolutions for the design pattern instances and provide a framework where each applied design pattern instance in the software system design can be transformed according to the properties of the pattern.

A Role-Based Meta-modeling Language (RBML) has been proposed to specify the design patterns in [23]. Based on RBML, a tool is also developed to check the conformance of a software system design diagram to a design pattern diagram by matching the pattern roles to the classes in the software system design diagram. If a software system design diagram does not conform to the corresponding pattern diagram, their tool is able to change the non-conformance case to a conformance one by adding missing roles into the software system design diagram. Although their approach can add missing roles of a pattern, the goal of their tool is to change the software system design to conform a pattern diagram, which is different from our goal of pattern evolutions. Thus, their tool can only add missing roles of a pattern, instead of a new pattern instance. In addition, our approach is based on QVT that follows model transformation standards.

With the intent to incorporate patterns into UML models, Kim *et al.* propose a role based meta-modeling approach to specify design patterns at the meta-model level [22]. They propose a notion of model role, which is generalized from the object-oriented role. Model role is used to describe the role played by a model element. This notion can be used to define the interactions and state-based behaviors of design patterns. They illustrate the use of model roles with a specification of a variant of the Observer design pattern. In contrast, we specify design patterns using predicates. In addition to the pattern specification, our work also includes the model transformation based approach for design pattern evolution.

Noda *et al.* [30] consider design patterns as a concern that is separate from the application core concern. Thus, an application class may assume a role in a design pattern by weaving the design pattern concern into the application class using Hyper/J [36]. Due to the separation of concerns, an application class may assume different roles in different design patterns. The change of roles that an application class plays, i.e., the change of design patterns, becomes a relatively simple task. The main goal of their evolution of design pattern is the replacement of one pattern by another. In contrast,

our design pattern evolution refers to the internal changes of a design pattern application. In addition, the practical application of their approach is left as a mystery.

Improving software system quality by applying design patterns in existing software systems has been discussed in [5][6]. When the user selects a design pattern to be applied in a chosen location of a software system, automated application is supported by applying transformations corresponding to the mini-patterns. The transformations are defined in terms of preconditions, algorithm, behavioral preservation, and postconditions. The main goal of their software evolution is to apply design patterns in existing software systems, whereas our evolution goal is to change the design patterns that have already been applied in a software system. Additionally, our transformations are defined at the UML meta-model level and implemented in QVT.

The tool support for UML model evolution is provided in [21], which is also based on XMI. The tool is mainly for database model scheme evolutions, such as the change of the type of a data element. The design and development of the tool applies several design patterns to model the components (Composite pattern), to reconcile difference in changing interfaces (Adapter pattern), to model a set of update commands (Composite pattern), and to describe change executions (Command pattern). While they apply some design patterns in their tool, they have not addressed the changes of a design pattern instance. In contrast, we focus on the evolution of design patterns, instead of the evolution of UML models.

Mazon *et al*. [27] applied MDA to a data warehouse (DW) development framework to address the design of the whole DW. The framework consists of five layers, each of which represents a part of the whole DW system. Each layer has its own viewpoint, including the Platform Independent Model (PIM), Platform Specific Model (PSM) and Computation Independent Model (CIM), according to the MDA. Each layer and its viewpoints require different model formalisms. Once PIM models in each layer are developed, other models can be transformed based on the relational layer of QVT. While they proposed QVT-based model transformations for DW, we presented QVT-based model transformations for design pattern evolutions. In contrast to their vertical transformations where a source model is transformed into a target one at different level of abstraction, our model transformations merge multiple source models into a single target model at the same level of abstraction. In addition, their model transformations are mostly presented at the conceptual level and lack tool support.

D'Ambrogio [8] applied model transformations to build automatically 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. While their goal of model transformation is to construct a performance model from the UML model, our goal is to automatically evolve design pattern applications. Moreover, their tool was implemented in XSLT, instead of QVT.

Kalnins *et al*. [19][20] 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 [41], ATL [34], MTF [38], Tefkat [42], and Fujaba Story diagrams (SDM) [35], which are either textual or graphical languages, address the transformation from one model to another. Muller *et al.* [28] 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.

XMI is an interchange format that supports the exchange of models in differential forms. Kovse and Härder [25] build on this feature of XMI to examine the possibility of XMI-based generic transformations of UML models. They propose to predefine generic XMI-based transformations, which can be configured via parameter values to generate specialized transformations via XSLT. The specialized transformations can be used to transform UML models. Therefore, their approach promotes model reuse. On the other hand, our work is more specific to the transformation of UML models. We provide the transformation framework to transform the design pattern instances applied in the software UML design model.

Lengyel *et al*. [26] investigate how to support flexible, expressive, and verifiable model transformation. They present a meta-model-based model transformation framework, Visual Modeling and Transformation System. They use graphs to visualize the models and perform the graph transformation that is facilitated by the graph rewriting tool. In addition to visualizing models by graphs, they use OCL to define the transformation rules precisely and to validate the model transformation. Meta-model-based transformation rules are at the layer of meta-models, which allows assigning OCL constraints to transformation rule elements. They provide a high-level control flow language, Visual Control Flow Language (VCFL), to define transformations. Although our approach also bases on model transformation, we deal with a different topic, which is the evolution of design pattern instances applied in the software system design.

8 Conclusions

Since the evolution information of a design pattern is generally implicit in the pattern descriptions, a designer has to dig into the pattern descriptions to understand the particular ways of evolution 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 in [11][13], many of the evolution processes recur in different patterns, which can be documented by studying the pattern descriptions.

In this paper, we define the evolution processes of design patterns as model transformation rules based on QVT. We specify design pattern properties and evolutions in predicates. We also provide tool support for the automation of the pattern evolution 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. To illustrate our approach, we also provide a case study on an open-source software system. The case study shows that the types of evolutions of each design pattern we define appear in real-life systems. Although we only use the Adapter pattern as the example and case study in this paper, our approach could be applied to other patterns as well. For example, we have presented an example of the Observer pattern evolution using our approach in [14].

9 References

[1] P. Alencar, D. Cowan, J. Dong, and C. Lucena, A Pattern-Based Approach to Structural Design Composition, *Proceedings of the IEEE 23rd Annual International Computer Software & Applications Conference,* pp160-165, Phoenix USA, October 1999.

[2] G. Antoniol, R. Fiutem, and L. Cristoforetti, Design pattern recovery in object-oriented software. *Proceedings of the 6th IEEE International Workshop on Program Comprehension (IWPC),* pp153-160, Ischia, Italy, June 1998.

[3] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide, Addison-Wesley, 1999.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern Oriented Software Architecture: A System of Patterns. John Wiley & Son. 1996.

[5] M.Ó. Cinnéide, 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.

[6] M.Ó. Cinnéide, P. Nixon, A methodology for the automated introduction of design patterns. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM),* pp463-472, Oxford, England, September 1999.

[7] W. F. Clocksin and C.S. Mellish. Programming in Prolog. Berlin: Springer-Verlag, 1987.

[8] A. D'Ambrogio, A Model Transformation Framework for the Automated Building of Performance Model from UML Models. *Proceeding of the 5th International Workshop on Software and Performance*, pp75-86, Palma, Illes Balears, Spain, July 2005.

[9] J. Dong, D.S. Lad and Y. Zhao, DP-Miner: Design Pattern Discovery Using Matrix. *Proceedings of the Fourteenth Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS),* pp371-380, Arizona, USA, March 2007.

[10] J. Dong, S. Yang and K. Zhang, Visualizing Design Patterns in Their Applications and Compositions, *IEEE Transaction on Software Engineering (TSE)*, Volume 33, Number 7, pp. 433-453, July 2007.

[11] J. Dong, S. Yang and K. Zhang, A Model Transformation Approach for Design Pattern Evolutions. *Proceedings of the Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS)*, pp 80-89, Potsdam, Germany, March 2006.

[12] J. Dong, S. Yang and D.T. Huynh, Evolving Design Patterns Based on Model Transformation. *Proceedings of the 9th IASTED International Conference on Software Engineering and Applications (SEA),* pp 344-350, Phoenix, USA, November 2005.

[13] J. Dong, S. Yang, and Y. Sun, A Classification of Design Pattern Evolutions. *International Journal of Object Technology (JOT)*, Volume 6, Number 10, pp95-109, November-December 2007.

[14] J. Dong, S. Yang, Y. Sun, and W.E. Wong, QVT Based Model Transformation for Design Pattern Evolutions. *Proceedings of the Tenth IASTED International Conference on Internet and Multimedia Systems and Applications (IMSA),* pp16-22, USA, Honolulu, Hawaii, USA, August 2006.

[15] J. Dong, Y. Zhao, and Y. Sun, A Matrix-Based Approach to Recovering Design Patterns. *IEEE Transactions on Systems, Man, and Cybernetics (TSMC), Part A,* pp1271-1282, November 2009.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[17] Y. Gueheneuc, H. Sahraoui, and F. Zaidi, Fingerprinting design patterns. *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE)*, pp172-181, Delft, Netherlands, November 2004.

[18] S.R. Judson, R.B. France, and D.L. Carver, Specifying Model Transformations at the Meta-model Level, *Proceedings of the Workshop in Software Model Engineering (WiSME@UML'2003)*, San Francisco UAS, October 2003.

[19] A. Kalnins, J. Barzdins, and E. Celms, Model Transformation Language MOLA. *Proceedings of the Second European Conference on Model-Driven Architecture: Foundations and Applications (MDAFA)*, pp. 14-28, Linkoeping, Sweden, June 2004

[20] A. Kalnins, J. Barzdins, and E. Celms, Model Transformation Language MOLA: Extended Patterns. *Proceedings of the 6th International Baltic Conference DB@IS 2004,* IOS Press, FAIA vol. 118, pp. 169-184, 2005.

[21] F. Keienburg and A. Rausch, Using XML/XMI for tool Supported Evolution of UML Models, *Proceedings of International Conference Hawaii International Conference on System Science*, pp3-6, Maui, Hawaii, January 2001.

[22] D.-K. Kim, R. France, S. Ghosh, and E. Song. A Role-based Meta-modeling Approach to Specifying Design Patterns. *Proceedings of the 27th IEEE Annual International Computer Software and Applications Conference (COMPSAC)*, pp452-457, Dallas, USA, November 2003.

[23] D.-K. Kim and W. Shen, An approach to evaluating structural pattern conformance of UML models. *Proceedings of ACM Symposium on Applied Computing (SAC)*, pp1404-1408, Seoul, Korea, March 2007.

[24] T. Kobayashi and M. Saeki. Software Development Based on Software Pattern Evolution, *Proceedings of the 6th Asia-Pacific Software Engineering Conference (APSEC)*, pp18-25, Takamatsu, Japan, December 1999.

[25] J. Kovse and T. Härder, Generic XMI-Based UML Model Transformations. *Proceedings of the International Conference on Object-Oriented Information Systems*, Montpellier: Springer-Verlag, pp192-198, 2002.

[26] L. Lengyel, T. Levendovszky, G. Mezei, T. Vajk and H. Charaf, Practical Uses of Validated Model Transformation. *Proceedings of the International Conference on Computer as a Tool*, pp 2200-2207, Warsaw, Poland, September 2007.

[27] J.-N. Mazon, and J. Trujillo, Applying MDA to the Development of Data Warehouses. *Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP*, pp 57-66, Bremen, Germany, November 2005.

[28] P.-A. Muller, F. Fleurey, D. Vojtisek, Z. Drey, D. Pollet, F. Fondement, P. Studer, and J.-M. Jezequel, On Executable Meta-Languages Applied to Model Transformations. *Proceedings of INRIA Workshop of Model Transformations In Practice*, Montego Bay, Jamaica, October 2005.

[29] J. Niere, W. Schafer, J. P. Wadsack, L. Wendehals, and J. Welsh, Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pp 338-348, Orlando, SUA, May 2002.

[30] N. Noda and T. Kishi, Design pattern concerns for software evolution. *Proceedings of the 4th International Workshop on Principles of Software Evolution,* pp 158-161, Vienna, Austria, September 2001.

[31] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis, Design Pattern Detection Using Similarity Scoring. *IEEE transaction on software engineering*, Vol. 32, No. 11, pp896-909, November 2006.

[32] X.-. Wang, Q.-Y. Wu, H.-M. Wang, D.-X. Sh, Research and Implementation of Design Pattern-Oriented Model Transformation. *Proceedings of the International Multi-Conference on Computing in the Global Information Technology*, pp24-29, French Caribbean, March, 2007.

[33] J.B. Warmer and A.G. Kleppe. The Object Constraint Language: Precise Modeling with UML. Addison-Wesley, 1998.

[34] ATL, http://www.sciences.univ-nantes.fr/lina/atl/

[35] Fujaba User Documentation, http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf

[36] Hyper/J, http://www.alphaworks.ibm.com/tech/hyperj

[37] Java.awt resource information, September 2006, http://java.sun.com/j2se/1.5.0/docs/guide/awt/index.html.

[38] MTF, http://www.alphaworks.ibm.com/tech/mtf

[39] Model Driven Architecture. http://www.omg.org/mda/

[40] OMG QVT specification, http://www.omg.org/docs/ptc/05-11-01.pdf

[41] QVT-Merge, http://www.omg.org/docs/ad/05-03-02.pdf

[42] Tefkat, http://www.dstc.edu.au/Research/Projects /Pegamento/tefkat/

[43] W3C, Extensible Markup Language (XML), http://www.w3.org/

[44] W3C, XSL Transformations (XSLT), http://www.w3.org/