
XSLT-based evolutions and analyses of design patterns



Jing Dong^{1,*}, †, Yajing Zhao¹ and Yongtao Sun²

¹*Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, U.S.A.*

²*American Airlines, 4333 Amon Carter Blvd, Fort Worth, TX 76155, U.S.A.*

SUMMARY

Design patterns document flexible designs that may evolve over time. Design pattern evolution typically involves the addition or removal of a group of modeling 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. Pattern instances may interact with each other making the evolution of design patterns more error-prone. Undetected design errors and inconsistencies may cause failures of systems. In this paper, we propose a service-oriented architecture for design pattern evolution and analysis based on two-level transformations, thus making the possible evolutions of each design pattern explicit. In addition, we automate the evolution processes as XSLT transformations that can transform the unified modeling language (UML) model of a design pattern application into the evolved UML model of the pattern. Both the original and evolved UML models are represented in the XML Metadata Interchange format to facilitate the transformations. Furthermore, we check the consistency of the evolution results using the semantic web checker based on the Java Theorem Prover. A case study on a large real-world system is presented to illustrate and evaluate our approach. Copyright © 2009 John Wiley & Sons, Ltd.

Received 4 February 2008; Revised 13 November 2008; Accepted 18 November 2008

KEY WORDS: design pattern evolution; model transformation; semantic web; UML; XSLT; XMI; SOA

1. INTRODUCTION

Software systems are normally required to be flexible to changes due to constant changes in user requirements, platforms, technologies, and environments. Change is a constant theme of software design and development. It can be a disaster if a single change causes a large number of changes in the systems. It is important to localize the changes such that minimum efforts are needed. This

*Correspondence to: Jing Dong, Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, U.S.A.

†E-mail: jdong@utdallas.edu

requires the initial designers of a software system to be aware of potential changes. Thus, the resulting systems are flexible and agile to future evolution.

One of the important goals of design patterns [1] is to design for change. Design patterns capture expert design experience by partitioning software designs into stable part and changeable part. By separating and encapsulating both parts, the change impact of a software design can be minimized. Thus, most of the design patterns encapsulate future changes that may only affect limited part of a design pattern. This evolution process can be achieved by adding or removing design elements in existing design patterns. In the 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 patterns, 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. Automation of these evolution processes can reduce errors and missing parts. In addition, the evolution of design patterns may interact with other pattern instances in the design. Such interactions may cause errors or inconsistencies among the pattern instances in a design. It is important to analyze the system design after evolutions to ensure the essential properties of pattern instances in the design.

To raise the level of abstraction, the model-driven architecture (MDA) [2] supports software development based on models as primary artifacts. Thus, the level of reuse is raised accordingly since high-level software models can be reused as well as software programs (libraries). In this way, models become assets in MDA. Consequently, technology that supports the transformation of models is considered as a key enabler of MDA. Design patterns are usually modeled in the unified modeling language (UML) [3], which is considered to be the *de facto* standard for object-oriented modeling. The evolution of a pattern may be considered as a transformation of the design model.

The XML Metadata Interchange (XMI) [4] is an interchange format for metadata in terms of the MetaObject Facility (MOF) [2]. XMI specifies how UML models are mapped into an XML file. By representing a UML model in the XML format, the UML model can be manipulated since there are rich collections of XML-related techniques and tools available. The extensible stylesheet language transformation (XSLT) [5] provides the transformation from an XML document to other types of documents (including XML). The use of XMI and XSLT helps to automate model transformation process and enforces constraints of model implicitly.

Services computing is a new paradigm that partitions the systems into three entities: service producer, consumer, and publisher. The service producer provides the functionalities that can be used by others. The service consumer is the application builder which requests the services provided by the service producer. The service publisher is a broker that registers the available services and allows the service consumer to discover the requested services. The service producer, consumer, and publisher are independent but collaborative through well-defined interfaces. In Services Computing, the service-oriented architecture (SOA) is an architectural style whose objective is to reduce coupling among interacting software agents. A service is a unit of work provided by a service provider to achieve desired results for a service consumer. Both provider and consumer are roles played by software agents. SOA can achieve loose coupling among interacting software agents by employing two architectural constraints. First, all participating software agents have a small set of simple and

ubiquitous interfaces. Only generic semantics are encoded at the interfaces. The interfaces should be universally available for all providers and consumers. Second, descriptive messages constrained by an extensible schema are delivered through the interfaces. No system behavior is prescribed by messages. A schema limits the vocabulary and structure of messages. An extensible schema allows new versions of services to be introduced without breaking existing services.

In this paper, we propose an SOA for design pattern evolution and analysis based on two-level transformations. To capture the evolution processes of design patterns, we define evolutions in two levels: the primitive level and the pattern level. The primitive-level evolutions are the addition or removal of modeling elements, such as classes and relationships. The pattern-level evolutions characterize the recurring evolutions of each design pattern based on the primitive-level evolutions. Meanwhile, the evolution processes are defined in the XMI format, describing how to transform the UML models of design pattern applications. Thus, the structure of a design pattern may be evolved in some prescribed ways based on the pattern-level transformation. The evolution processes in the XMI format are later translated into XSLT transformation rules. We provide services that can change the pattern-based design by adding or removing design elements in existing design pattern applications to automate pattern evolutions. In order to assure that the essential properties of a design pattern still hold after the evolutions, we develop a semantic web checker based on the Java Theorem Prover (JTP) [6,7] to check the consistency of the designs after the evolution processes. To illustrate and evaluate our approach, we present a case study on a real-world system with a design that has several design pattern instances. We change the system by performing certain evolutions on existing pattern instances and analyze the evolved system by checking several pattern properties.

The remainder of this paper is organized as follows: the following section describes the SOA including web-based pattern evolver, XMI2RDF converter, and semantic web consistency checker. Section 3 presents the details of the pattern evolver that can assist design pattern evolutions. Section 4 details the XMI2RDF converter and semantic web checker that help to analyze the properties of patterns and designs. Section 5 discusses a case study on a real-world system to illustrate and evaluate our approach. The last two sections cover related work and conclusions.

2. MOTIVATION AND OVERALL ARCHITECTURE OF OUR APPROACH

Design patterns [1] have become the design language for software teams to communicate design decisions and models. An important goal of design patterns is to design for change. Applying design patterns makes the system easy to maintain/change. There are changes that are related to the existing design pattern instances as well as those that are not. For example, Figure 1(a) shows a Mediator pattern instance, including the Mediator, ConcreteMediator, Colleague, ConcreteColleague1, and ConcreteColleague2 classes. One possible change of the system is to add some classes that do not have any relationship with the classes in the Mediator pattern instance. The new classes do not play any role in the Mediator instance. This change is not related to this Mediator pattern instance. Another possible change of the design is to add a new subclass of Colleague that participates in the existing Mediator pattern instance, as illustrated in Figure 1(b). The new class plays the role of ConcreteColleague. This change is compliant with the intent of the Mediator pattern that is designed to ease the future changes and evolutions when new concrete colleagues are added. In this paper, we focus on the second kind of changes because they describe the possible future evolutions

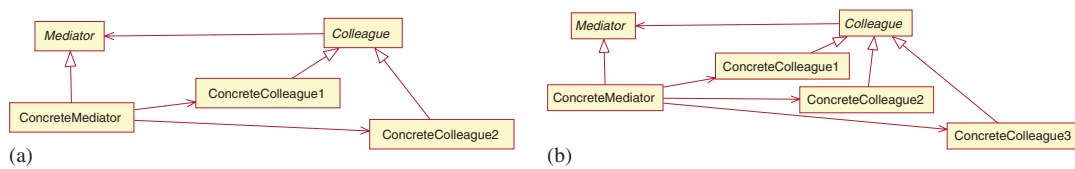


Figure 1. Mediator pattern and its possible evolution.

of the design patterns. Therefore, such changes are typically restricted by the corresponding design patterns and should not violate the constraints and properties of design patterns. On the other hand, the first kind of changes has nothing to do with the patterns applied in a system. These arbitrary changes are common in object-oriented designs, which is out of the scope of this paper.

The objective of our approach is to classify pattern evolutions, automate the evolution process, and ensure the consistency. It is less costly to perform evolution of a software system at the design level than at the implementation level. Model transformation is one of the important parts of the MDA, which allows software design model to be transformed to a new model at the same level or at different level, e.g. implementation level.

Design errors and inconsistencies are difficult to find and correct when they are transformed into implementation errors. Therefore, it is important to allow system design changes and error detections at the design level, instead of the implementation level.

Figure 2 depicts the SOA of our approach that consists of three components: the web-based pattern evolver, XMI2RDF converter, and semantic web consistency checker. The first component allows the developers to change the pattern-based design by adding or removing a group of modeling elements according to the pattern evolution specifications. The second component allows the developers to convert the evolved design from the XMI format to the resource description framework (RDF) format. The third component can check the consistencies of evolutions using our semantic web checker. In the remainder of this paper, we will describe these three components in details.

3. WEB-BASED PATTERN EVOLVER

The main goal of the web-based pattern evolver is to provide a service that can select some candidate evolutions from an evolution library and apply these evolutions to the UML design model based on the corresponding XSLT transformation rules [5,8].

3.1. Evolution library

Most design patterns are designed to allow certain future changes. When a design pattern is applied, it leaves space for flexible changes in the future. Such changes may include the addition or removal of certain classes, attributes, and operations, which have minor impact of the whole system design. However, a design pattern is typically defined in the form of an essay [1], with headings such as intent, motivation, structure, behavior, applicability, consequence, etc. The possible changes of a design pattern are normally embedded implicitly in the description of the pattern. The designers

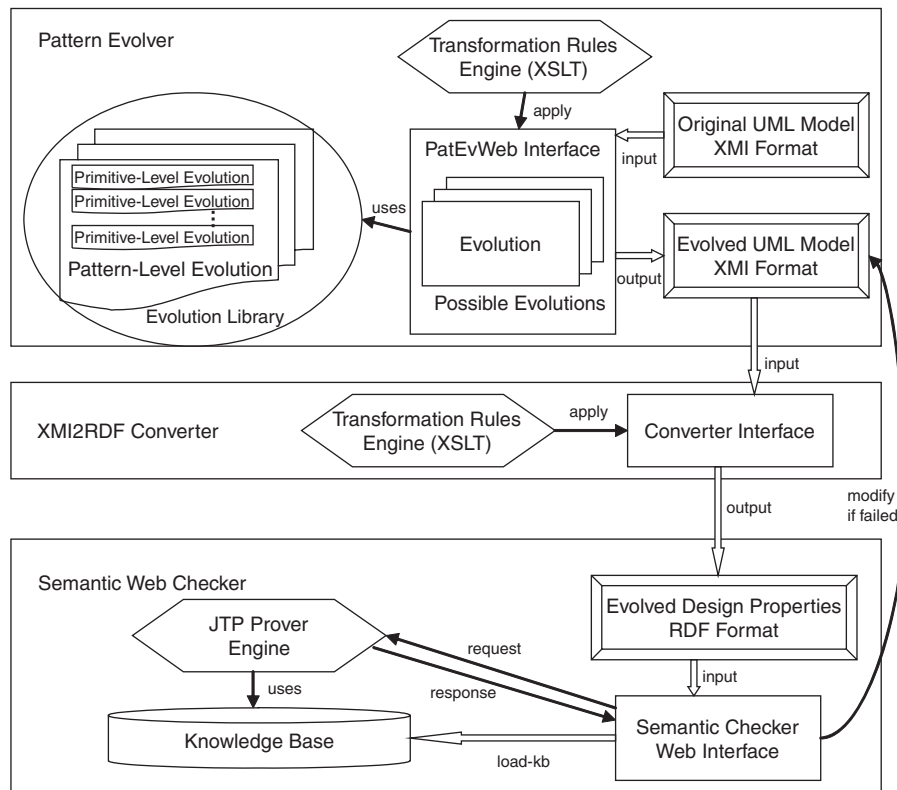


Figure 2. The overall architecture of the approach.

have to read between the lines of the pattern description when changes are needed. Thus, it is important to explicitly capture the possible changes into predefined pattern evolutions.

To classify the possible pattern evolutions, we analyze the design patterns documented in [1] that are carefully selected by GoF. These patterns are representative design experience. Although various design patterns have later been documented in the literature, most of them still follow GoF style. In this paper, we categorize the pattern evolutions into an evolution library based on the representative design patterns in [1]. In particular, we analyze the evolutions and decompose them into lower-level evolutions, which we call primitive-level evolutions [8]. The primitive-level evolutions are the addition or removal of modeling elements, such as classes and relationships. We have identified a total of nine modeling elements that can be added or deleted as the primitive-level evolutions. On the other hand, the previously categorized evolutions are called pattern-level evolutions, which can be described in terms of a sequence/combination of the primitive-level evolutions. We also provide a schema to classify these pattern-level evolutions into several categories. Instead of a complete list of pattern evolutions, our main goal is to provide a method to classify them and automate the pattern evolutions as model transformation. New pattern evolutions can be defined with the primitive-level evolutions.

Table I. The primitive-level evolutions.

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

3.1.1. Primitive-level evolutions

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

We identify nine modeling elements that can be added or deleted as the basic transformations in the pattern evolution process. The general format of adding a modeling element is Add (ME (PL)). The model elements (ME) and the parameter list (PL) are shown in Table I. For example, adding a class named 'Leaf' can be specified: Add (Class (Leaf)). Similarly, the removal of a modeling element can be specified: Delete (ME (PL)). The replacement of an ME with another is conducted by first removing the modeling element and then adding a new modeling element. It can be defined: Delete (ME1(PL1)) + Add(ME2(PL2)).

3.1.2. Pattern-level evolutions

The pattern-level evolutions characterize the possible evolutions of each design pattern. They are described in terms of a sequence/combination of the basic primitive-level evolutions. In this section, we categorize the pattern-level evolutions into five categories that are recurring in different design patterns based on our previous work [8]. Note that we do not claim this to be a complete list of pattern-level evolution categories. By granulating the evolutions into lower level and categorizing

Table II. Categories of pattern-level evolution.

#	Evolution names	Description
1	Independent	Addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern
2	Packaged	Addition or removal of one independent class with attributes and/or operations and the corresponding relationships between this class and the classes in the original pattern
3	Class group	Addition or removal of one attribute/operation in several different classes consistently
4	Correlated classes	Addition or removal of a group of correlated classes
5	Correlated attributes/operations	Addition or removal of a group of classes and addition or removal of some attributes or operations in the classes of the original pattern applications

them, however, our approach is extensible when new pattern evolutions are identified. They can be specified by the combinations of the primitive-level evolutions and thus supported by our tool. In addition, we concentrate on classifying the structural evolutions, instead of behavioral ones. The evolutions at structural level often affect the behavior of the design. The addition or removal of design parts may correlate to the behavior of corresponding objects. Thus, structural evolutions typically result in behavior changes. The behavioral changes without corresponding structural evolutions are out of the scope of this paper because they may not relate to the evolutions of design patterns.

Table II lists the five categories of pattern-level evolutions. In the remainder of this section, each pattern-level evolution category is explained in detail.

The first pattern-level evolution is called *independent* change which is a simple addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern. This class is independent in the sense that the addition or removal of the class does not cause any effects on the existing classes of the design. This kind of pattern-level evolution can be expressed in the primitive level evolutions as follows[‡]:

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

where the `className` is the name of the class which is to be added into the pattern. The `Relationship` includes association, generalization, aggregation, composition, realization, and dependency. The `existingClassName` is the name of the class from the original pattern. There may be multiple relationships added into the pattern with the addition of a class. This kind of evolution appears in several design patterns, for example, in the Mediator and Facade patterns. Figure 1(a) is the class diagram describing a possible application of the Mediator pattern containing two `ConcreteColleague`

[‡]Since the addition and removal have the same format and the only difference is the transformation names (Add and Delete), we omit the evolutions of removing modeling elements. We also omit the specifications of the next four kinds of evolutions. We refer to [9] for interested readers.

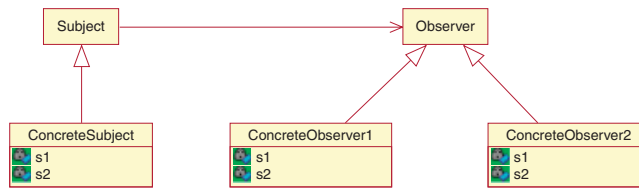


Figure 3. Observer pattern with two concrete observers.

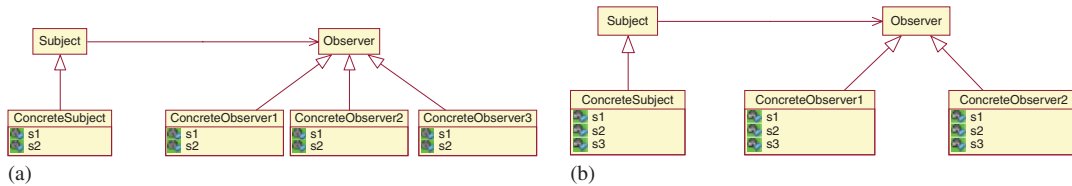


Figure 4. Two possible evolutions for observer pattern.

classes. A potential evolution of this pattern application is to add a new ConcreteColleague class, which can be defined as the following transformations based on the primitive-level evolutions:

```

Add (Class (ConcreteColleague3)) +
Add (Generalization (ConcreteColleague3, Colleague)) +
Add (Dependency (ConcreteMediator, ConcreteColleague3))
  
```

where a new ConcreteColleague class is added along with two new relationships: generalization and dependency. The generalization relationship is with the Colleague class. The dependency relationship is on the ConcreteMediator class. The result of this evolution is shown in Figure 1(b).

The second pattern-level evolution is called *packaged* change which is the addition or removal of one independent class and the corresponding relationships between this class and the classes in the original pattern. In addition, certain attributes and/or operations of this class are added and removed accordingly. For example, Figure 3 is the class diagram describing a possible application of the Observer pattern containing one ConcreteSubject and two ConcreteObserver classes. A potential *packaged* evolution of this pattern application is to add a new ConcreteObserver class (ConcreteObserver3) with its attributes (s1 and s2) as shown in Figure 4(a).

The third kind of pattern-level evolution is called *class group* change which 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. One potential class group evolution of the Observer example in Figure 3 is to add one attribute called s3 as a new data to be observed by the observers. Thus, this attribute needs to be added in all ConcreteSubject and ConcreteObserver classes, which is shown in Figure 4(b).

The fourth kind of pattern-level evolution is called *correlated classes* change which is the addition or removal of a group of correlated classes. When certain classes are added or removed, some other classes have to be added or removed accordingly. These correspondence relations are important since missing transformations may cause inconsistency. In addition, the corresponding relationships

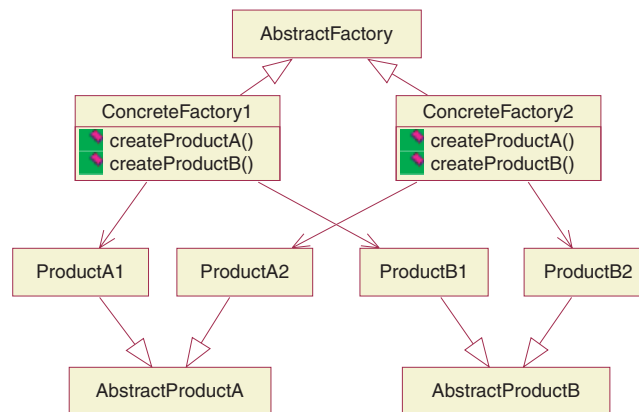


Figure 5. Abstract factory pattern with two kinds of products created by two concrete factories.

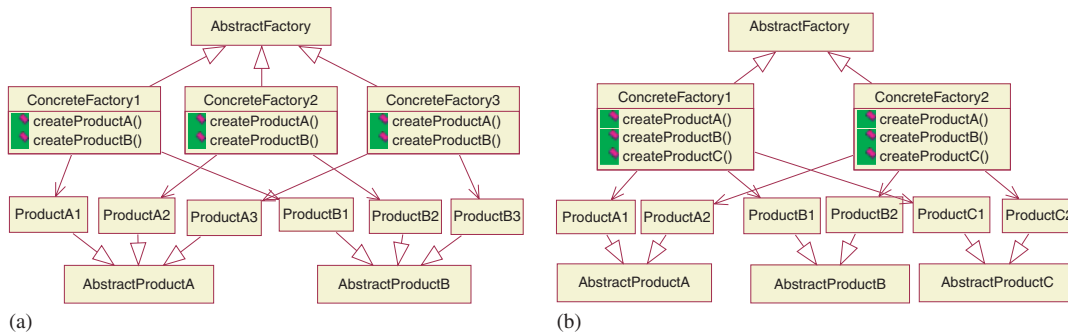


Figure 6. Two possible evolutions for abstract factory pattern.

between this group of classes and other classes are added or removed. The attributes and operations of the classes of this group are also added or removed. The addition or removal of this group of classes may not affect the internal of other classes in the original design pattern applications. For example, Figure 5 shows an application of the Abstract Factory pattern with two kinds of products (AbstractProductA and AbstractProductB). Each kind of products has two concrete products: ProductA1/ProductB1 and ProductA2/ProductB2, respectively. Thus, there are two concrete factories: ConcreteFactory1 and ConcreteFactory2. A potential correlated-classes evolution can be the addition of a new family of concrete products (ProductA3 and ProductB3). This requires the addition of a new concrete factory (ConcreteFactory3) to create the corresponding newly added concrete products. This new concrete factory class also has the same operations (createProductA and createProductB) as the other two concrete factory classes as shown in Figure 6(a).

The fifth kind of pattern-level evolution is called *correlated attributes/operations* change, which is the addition or removal of a group of classes. This change also requires the addition or removal of some attributes or operations in the classes of the original pattern applications. For the same example shown in Figure 5, a potential correlated attributes/operations evolution can be the addition

of a new kind of product (AbstractProductC with ProductC1 and ProductC2). This requires the addition of the createProductC operation in all concrete factory classes (ConcreteFactory1 and ConcreteFactory2). The corresponding generalization and realization relationships are also added as shown in Figure 6(b).

3.1.3. Categorization of pattern evolutions

We studied the pattern evolutions for the design patterns listed in [1]. Each design pattern may perform several pattern-level evolutions, which can be added in the documentation of the pattern. We define the possible evolutions of each design pattern in an XML file as a library that provides the candidate evolutions for our pattern evolver service. For instance, the Adapter pattern has two possible pattern-level evolutions: adding a new adapter into the pattern and adding a new Adaptee into the pattern. For adding a new Adaptee evolution, a class that plays the role of Adaptee in the Adapter pattern is added into the pattern instance. Meanwhile, a class that plays the role of Adapter in the Adapter pattern should be added simultaneously. The Adaptee class should define a SpecificRequest operation and the Adapter class should define a Request operation. If the new class name is the same as existing class name that plays the role of Adapter or Adaptee, the class will be reused by the new pattern instance. Otherwise, a new class will be created and included in the new pattern instance. Similarly, if the Request or the SpecificRequest operations already exist, it will be included in the pattern instance; otherwise, new operation will be added and included in the pattern instance.

There are different types of evolutions for each pattern. For example, one possible evolution of the Abstract Factory pattern is shown in Figure 6(a), which is classified as the fourth type of pattern-level evolution (correlated classes in Table II). In this type of evolution, the addition of a new set of concrete products (ProductA3 and ProductB3) results in the addition of the ConcreteFactory3 class. The other possible evolution is the fifth type (correlated attributes/operations) of pattern-level evolution as, for example, depicted in Figure 6(b). The addition of a new set of concrete products (ProductC1 and ProductC2) results in the addition of AbstractProductC class and the addition of operation (createProductC()) in the existing classes, ConcreteFactory1, and ConcreteFactory2. Thus, the Abstract Factory pattern may have the fourth and fifth types of possible pattern-level evolutions. The application of the Singleton pattern is not typically intended to evolve, it has no evolution. Table III shows the number of each type of evolution we have identified for each design pattern.

3.2. XSLT-based transformation rule engine

In order to automatically evolve design patterns, the evolution processes are implemented in the XMI format to transform the UML models of design pattern applications. Thus, the structure of a design pattern may evolve in some prescribed ways according to the pattern-level transformation. The evolution rules of design patterns are specified as XSLT transformation rules based on the two-level evolutions. These transformation rules define that modeling elements are changed (added or removed) as well as where and how these elements are changed. We also provide tool support to automate such transformations. In our approach, every primitive-level evolution has a corresponding XSLT transformation rule which can add/remove the corresponding modeling element to the original UML model in the XMI format. Thus, the pattern-level evolution is a sequence

Table III. The evolutions of design patterns.

Design pattern name	Pattern-level evolutions
Abstract factory	4, 5
Builder	4, 5
Factory method	4
Prototype	2
Singleton	N/A
Adapter	4, 5
Bridge	2
Composite	2
Decorator	2, 3
Façade	1
Flyweight	2
Proxy	4
Chain of responsibility	2
Command	4
Interpreter	2
Iterator	4
Mediator	1
Memento	3
Observer	2, 3
State	2
Strategy	2
Template method	2, 3
Visitor	2, 5

of primitive-level evolutions. 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 are a group of XSLT transformation rules associated, which can add/remove the ME corresponding to this pattern-level evolution into/from the original UML model in the XMI format.

A software design with the applications of design patterns is normally modeled using the UML and drawn using UML tools, such as IBM Rational Rose [10] and ArgoUML [11]. As UML diagrams are typically saved in proprietary formats of the corresponding UML tools, a standard XMI format of the UML diagrams has been defined and the plug-in of these UML tools has been developed to export UML diagrams into the XMI format. For example, the plug-in of IBM Rational Rose is called UniSys which can translate UML diagrams into the XMI format.

3.2.1. Automating the pattern evolution by model transformation

In the previous section, we introduce two levels of evolutions: primitive level and pattern level. We also explicitly describe the evolution processes of several design patterns in terms of the two-level evolutions. In order to automate the evolution process, we use the XSLT processor to transform from one UML model of a design pattern into the evolved UML model of the design pattern. More specifically, we translate the UML model of a design pattern into the XMI format and describe our two-level evolutions as XSLT transformation rules. Using an XSLT processor, design pattern

Table IV. The primitive-level evolutions in XMI.

Evolutions	Subtags of <XMI.Add> and <XMI.Delete>
Class	<UML:Class name = '...' />
Attribute	<UML:Attribute attributeName = '...' className = '...' />
Operation	<UML:Operation operationName = '...' className = '...' />
Association	<UML:Association associationEnd1 = '...' associationEnd2 = '...' />
Generalization	<UML:Generalization child = '...' parent = '...' />
Aggregation	<UML:Aggregation wholeName = '...' partName = '...' />
Composition	<UML:Composition wholeName = '...' partName = '...' />
Realization	<UML:Realization fromName = '...' toName = '...' />
Dependency	<UML:Dependency fromName = '...' toName = '...' />

evolutions can be automated by transforming from the original UML model of a design pattern to the destination UML model of the pattern.

Although there are several model transformation languages available currently, such as MDR (MetaData Repository from Sun [12]) and EMF (Eclipse Modeling Framework from IBM [13]), we choose XSLT to be our modeling transformation language due to the following reasons. First, current modeling languages are mostly based on MOF QVT, which is not yet finally standardized. Each modeling transformation language interprets the QVT in a different way. Second, since we are dealing with the primitive-level transformation, which is the basis of higher-level transformations, i.e. pattern-level transformations, it is intuitive to use some basic type of model transformation language such as XSLT. Thus, we can define the semantic meaning of these primitive-level pattern transformations.

3.2.2. Primitive-level evolutions in XSLT

In this section, we describe the representations of the primitive-level evolution processes based on XSLT transformation rules. In particular, every primitive-level evolution shown in Table I can be represented in XMI by using XMI tags. More specifically, the addition or removal of a modeling element can be implemented in XMI by the tags '<XMI.Add> subtags </XMI.Add>' and '<XMI.Delete> subtags </XMI.Delete>', respectively. Table IV shows the subtags corresponding to each primitive-level evolution in Table I. For instance, the first primitive-level evolution Add(Class(className)) can be implemented in XMI as follows:

```
<XMI.Add>
  <UML:Class name = '...' />
</XMI.Add>
```

Therefore, suppose the new class name is 'ConcreteObserver3'. The first primitive-level evolution is implemented in the XML file 'New_Class.xml' in Figure 7(a) that also shows some other primitive-level evolutions, such as the additions of two attributes and one generalization relationship.

In our approach, every primitive-level evolution has a corresponding XSLT transformation rule which can add/remove the corresponding modeling element to the original UML model in XMI format. Thus, the pattern-level evolution is a sequence of primitive-level evolutions as, for example,

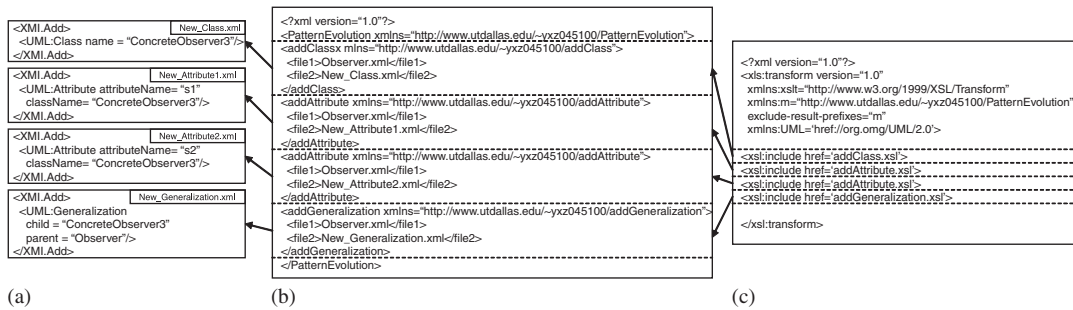


Figure 7. Pattern-level evolution of the observer pattern in XSLT: (a) primitive-level evolution; (b) pattern-level evolution; and (c) XSLT transformation rule.

defined in Figure 7(b), where the <addClass> tag describes the addition of a class by merging the original UML model (<file1>) and the new class (<file2>). Similarly, the <addAttribute> and <addGeneralization> tags describe the additions of an attribute and a generalization relationship, respectively.

3.2.3. Pattern-level evolutions in XSLT

The pattern-level evolutions are represented in XMI similarly by grouping the XMI specifications of the corresponding primitive-level evolutions. For example, the second kind of pattern-level evolution (packaged evolution) can be implemented in XMI as shown in Figure 7(b), where a new class named ‘ConcreteObserver3’ is added into the original application of the Observer pattern. Two attributes, s1 and s2 of the ConcreteObserver3 class, are also added. In addition, the generalization relationship between the ConcreteObserver3 and Observer classes is added with the ConcreteObserver3 class as a child and the Observer class as a parent. Every ME to be added is expressed in a separate XML file, e.g. the ‘ConcreteObserver3’ class is defined in the file named ‘New_Class.xml’, which corresponds to the primitive-level evolution of ‘add a class’ defined in Table IV.

Each pattern-level evolution associates with a group of XSLT transformation rules, which can add/remove the ME corresponding to this pattern-level evolution into/from the original UML model in the XMI format. Figure 7(c), for instance, shows the XSLT transformation rules of the packaged evolution for the Observer pattern, which adds four ME, a class, two attributes, and a generalization relationship, into the original application of the Observer pattern as shown in Figure 3. The evolution result is shown in Figure 4(a). The ‘addClass.xml’ file describes the transformation rules on how to add the new class (New_Class.xml) into the original Observer pattern application (Observer.xml). It locates these two XML files by looking for the <file1> and <file2> tags in the <addClass> tag from the pattern-level evolutions as, e.g. described in Figure 7(b). Similarly, the ‘addAttribute.xml’ and ‘addGeneralization.xml’ files describe the transformation rules for the additions of the attributes and generalization, respectively. These XSLT files can be reused in any corresponding transformation requests. For brevity, we omit the detailed contents of these XSLT files.

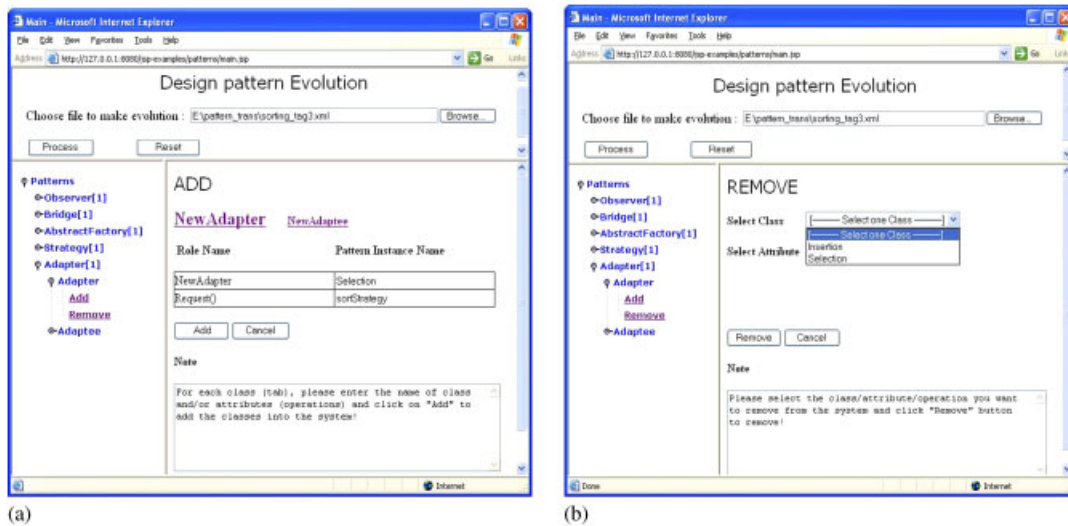


Figure 8. Pattern evolution automation system.

3.3. Web interface and UML model (XMI format)

We developed a web-based tool for the selection and invocation of design pattern evolutions. Two example screenshots of this tool are shown in Figure 8. Figure 8(a) shows the web interface in which the user can select the pattern-level evolutions to apply in the system design. The input of the tool is a system design in the XMI format, which can be generated by a UML tool and the corresponding plug-in (such as Rational Rose with the Unisys plug-in). The XMI format of the system design has the pattern-related information attached. After the file is uploaded through the web interface, all the patterns applied in the system are listed on the left panel of web interface in Figure 8(a). The sub-items of each design pattern in the pattern tree are the possible evolutions of the design pattern. The user can select the evolutions to apply to the system design.

For each evolution, there are two actions: add and remove. When the user chooses to add a (or a group of) modeling element(s), the right panel of the user interface collects all necessary information such as the name of the ME. The addition action is triggered by clicking on the 'Add' button.

When the user wants to remove modeling element(s) from the system, he/she can select 'remove' item from the pattern tree on the left panel of the user interface. Figure 8(b) shows the removal action of the Adapter pattern. When the 'remove' item is selected, the classes that participate in the Adapter pattern are displayed in a drop-down list on the right panel of the user interface. The user can select the class he/she wants to remove from the Adapter pattern and click on the 'Remove' button to remove the select class from the system.

4. SEMANTIC WEB CONSISTENCY CHECKER

When a software system evolves, the integrity and consistency of the system should be maintained. Some properties may not hold anymore after the changes and evolutions [14]. Thus, we need to

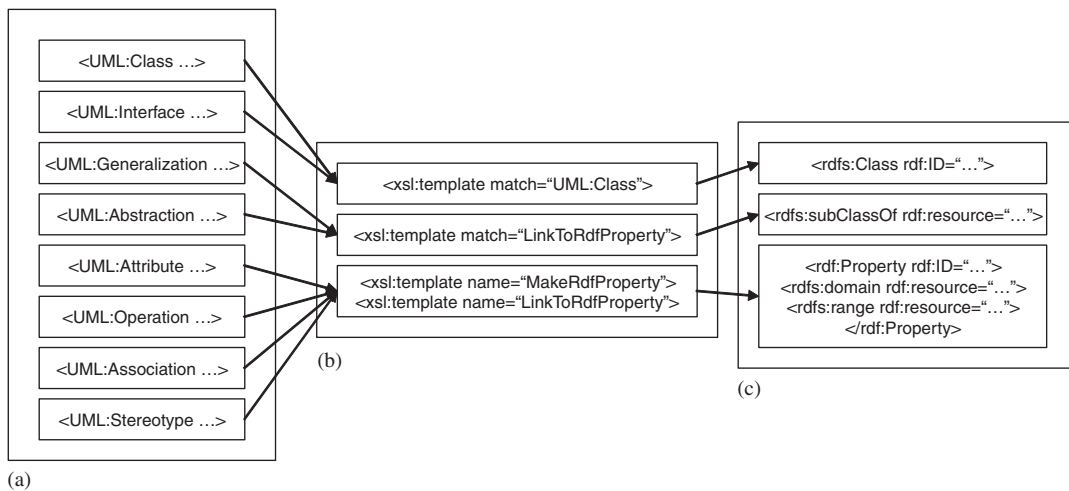


Figure 9. XSL transformation from XMI to RDF: (a) XMI format; (b) XSL transformation rule; and (c) RDF format.

check that the evolved system does not violate any pattern properties or any predefined system properties. In order to assure that the essential properties of a design pattern still hold after the evolutions, we provide a consistency checker service based on the JTP [6,7] to check the consistency of the designs after the evolution processes.

The main goal of the semantic web consistency checker is a service to check whether the important properties of design patterns hold in the knowledge base containing the design and evolution information based on a JTP engine, a knowledge base reasoning system. The knowledge base is represented in the RDF [15] format.

4.1. Transforming XMI to RDF knowledge base

In order to check the consistency of the evolved design, the evolved design in the XMI format is first converted into the RDF format, which can be used by the consistency checker to perform the system consistency checking. This conversion can be automated by our XMI2RDF converter adapted from [16].

The transformations in the converter can be generally depicted by Figure 9, where each `<UML:Class ...>` or `<UML:Interface ...>` tag in XMI is transformed into `<rdfs:Class ...>` tag in RDF, each `<UML:Generalization ...>` or `<UML:Abstraction ...>` tag in XMI is transformed into `<rdfs:subClassOf ...>` tag in RDF, and all attributes, operations, associations, pattern-related stereotypes, and so forth are transformed into `<rdf:Property ...>` tag by our XSL transformation template as shown in Figure 9.

4.2. Properties

After a system design evolves, we need to ensure that the evolved design persists the same structural and behavioral properties required by the original system (assuming that the original system design

has proper properties). There are two possible ways that the properties of the system are affected due to the system evolutions. First, the properties of design patterns applied in the system design no longer hold after the system evolution. When a new concrete product is added into the Abstract Factory pattern application, for example, the corresponding ConcreteFactory class should also be added into the pattern application. Otherwise, the properties of the Abstract Factory pattern no longer hold. Our approach can avoid these kind of problems by defining standard pattern-level evolutions for each design pattern. When the pattern application evolves, the user may choose a particular pattern-level evolution of the design pattern and apply our XSLT transformations, which include a group of tasks. For the Abstract Factory pattern example, the addition of a concrete product is grouped with the addition of the corresponding concrete factory such that the user may not miss either of them. Second, the evolution of a design may introduce inconsistencies into the system even though the properties of a certain pattern still hold. It may violate the properties of another pattern or introduce undesired properties. For example, the introduction of a group of classes and generalization relationships into a design may result in a circular generalization in the design, i.e. class A is a subclass of class B which is a subclass of class C which is a subclass of A. Our transformation rules do not prevent these kind of inconsistencies. Thus, we need to perform system inconsistency checking after the system evolution.

The properties that need to be checked can be defined based on the selected pattern-level evolutions and the UML model of the system design. The system properties are then expressed in JTP in term of queries. More complicated queries may have logical operators, such as 'and', 'or', 'not'. These system properties can be proved or disproved using the JTP reasoner. Figure 10 shows an example of the query in JTP, which tests whether there is circular inheritance in the design, i.e. the ConcreteSubject class is a subclass of the Subject class which is also a subclass of the ConcreteSubject class.

4.3. JTP prover engine

JTP prover engine performs the semantic search based on the knowledge base. JTP provides some built-in backward-chaining reasoner, which can accept proof goals and return answers together with the proofs of the answers. JTP Prover itself is extensible and the users can develop their own reasoner to achieve their specific goals. The input of the JTP prover is a query written in the RDF format. The outputs are matching results found by JTP prover based on the loaded knowledge base.

4.4. Checker web interface

At the client site, we provide a web interface for our semantic web consistency checker as shown in Figure 11. There are three main parts of the web interface: load KB, queries, and checking

```
(and (lhttp://www.w3.org/2000/01/rdf-schema#::lsubClassOf
lhttp://www.utdallas.edu/~yxz045100/observer.owl#::lConcreteSubject
lhttp://www.utdallas.edu/~yxz045100/observer.owl#::lSubject)
(lhttp://www.w3.org/2000/01/rdf-schema#::lsubClassOf
lhttp://www.utdallas.edu/~yxz045100/observer.owl#::lSubject
lhttp://www.utdallas.edu/~yxz045100/observer.owl#::lConcreteSubject))
```

Figure 10. Example Query in RDF/RDFS.

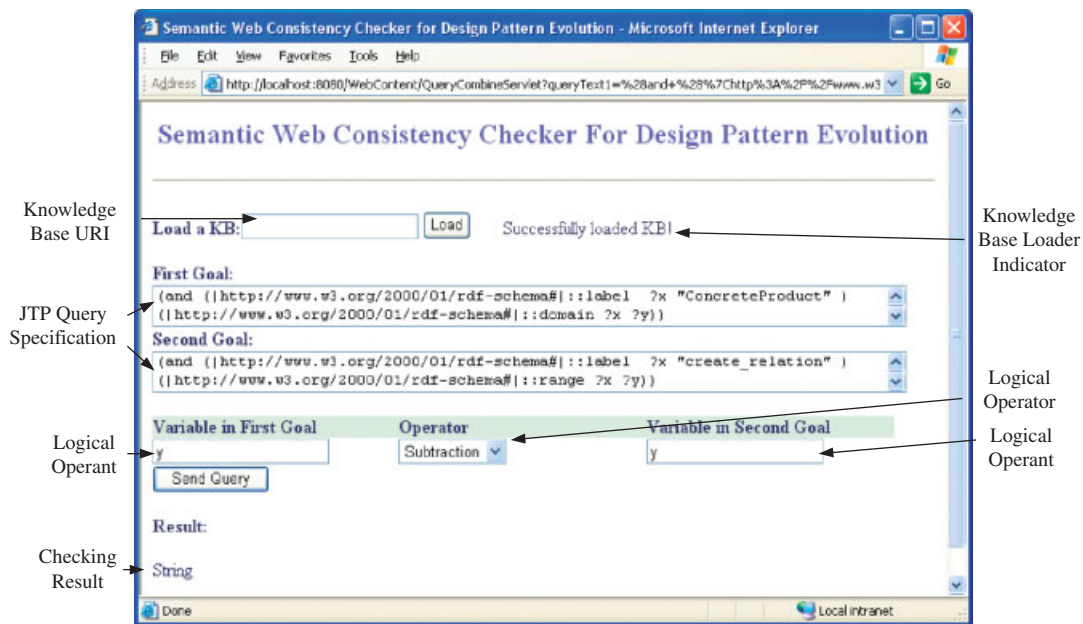


Figure 11. System consistency check Web interface.

result. First, the top panel allows the user to upload the knowledge base of the system design into the checker. As described previously, the knowledge base can be generated from the UML design model. In the second part, our checker allows the user to conduct two queries for two different goals (first goal and second goal) and the operations on the results of the two queries. The main reason why we provide this service is because JTP provides weak support on logic operations among queries. The user may conduct separate queries in JTP. However, the user has to manually compute the logic operations on the query results. In our checker service, the user is able to specify two different queries and select the logic operations, such as subtraction, union, and intersection, which can be applied on the results of the two queries. Our checker can automatically compute the checking result based on the queries. If the consistency checking only involves one query, the user can leave the second query and the logic operations blank. Our checker can provide the query result of the first goal. Finally, the checking result is reported to the user in the last part. If the result indicates that the property holds, the user may continue checking other properties. If the result indicates that the property does not hold, the user may analyze the design model in the light of the checking result to find the problem. After correcting the problem, the user may check the same property against the modified knowledge base.

5. CASE STUDY

In this section, we present a case study. The purposes of this case study are: (1) to demonstrate the feasibility of our approach; (2) to show how our Pattern Evolver automates the evolution process;

(3) to explain how our Pattern Evolver ensures the pattern integrity; and (4) to confirm the necessity of using our Semantic Web Checker to further detect inconsistencies and other system design mistakes.

The case study is conducted on a partial design of Java.AWT package in Java Development Kit (JDK) 1.4. AWT stands for Abstract Window Toolkit, a powerful and flexible Java class library for the development of Graphical User Interface (GUI). It contains rich user components like Frames and Panels which ease GUI development 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 like Scrollbar, Buttons, etc., onto containers like Frame or Panel. These layouts do not depend on window size or screen resolution [17].

Figure 12 shows a partial design of Java AWT package. For simplicity, some operations that are not involved in the discussion are not displayed in the diagram. The UML model inside the box of Figure 12 depicts the original design, which uses several design patterns, i.e. the Composite, Abstract Factory, Adapter, Strategy patterns.

To recursively design GUI widgets, such as button, canvas, and scrollbar, the Composite pattern is applied in this design. The Component and Container classes play the roles of the Component and Composite, respectively, whereas the Button, Canvas, Checkbox, Choice, Label, Scrollbar, and TextComponent classes play the role of Leaf.

There is one instance of the Abstract Factory pattern. The AbstractFactory role is played by the Paint class, whereas the ConcreteFactory is played by the Color, GradientPaint, TexturePaint classes. Within these classes, the createContext operation is the CreateProduct in the factory classes. Meanwhile, the AbstractProduct is played by the PaintContext class, and the ColorPaintContext, GradientPaintContext, TexturePaintContext classes play the role of ConcreteProduct.

There are three instances of the Adapter pattern in this design. One of them consists of the Component, Container, and LightweightDispatcher classes. The other contains the AWTEventListener, LightweightDispatcher, and Container classes. The third includes the BufferStrategy, BltBufferStrategy, SingleBufferStrategy, FlipBufferStrategy, and BufferCapabilities classes. Table V shows the detail information about the role that each class or operation plays in each Adapter pattern instance.

There are nine instances of the Strategy pattern in the design. Table VI shows the details about these nine instances, including the name of the participating classes, the role each class plays, the name of the operations that play the role of ContextInterface and AlgorithmInterface in the pattern instances.

Let us consider the evolution of this piece of the system design. For example, suppose the following four evolutions of the existing pattern instances in the original design are performed.

Table V. Adapter pattern instances in Figure 12.

Target	Adapter	Request	Adaptee	SpecificRequest
1 Component	Container	removeNotify	LightweightDispatcher	dispose
2 AWTEventListener	LightweightDispatcher	eventDispatched	Container	getMouseEventTarget
3 BufferStrategy	BltBufferStrategy SingleBufferStrategy FlipBufferStrategy	show	BufferCapabilities	getFlipContents

Table VI. Strategy pattern instances in Figure 12.

Context	Context/Interface	Strategy	ConcreteStrategy	Algorithm/Interface
1	Component	processComponentEvent	ComponentListener AWTEventMulticaster NativeInLightFixer	componentResized componentMoved componentShown componentHidden
2	Container	processContainerEvent	ContainerListener	componentAdded componentRemoved
3	Choice	processItemEvent	ItemListener	itemStateChanged
4	Checkbox	processItemEvent	ItemListener	itemStateChanged
5	Button	processActionEvent	ActionListener	actionPerformed
6	TextComponent	processTextEvent	TextListener	textValueChanged
7	Scrollbar	processAdjustmentEvent	AdjustmentListener	adjustmentValueChanged
8	Container	addNotify readObject writeObject	Component	addNotify readObject writeObject
9	Component	getBackBuffer	BufferStrategy FltpeBufferStrategy	getBackBuffer

1. *Composite Pattern Evolution.* In the Composite pattern instance, add a new Leaf, the MyComp class, which is a subclass of the Component class. As this new Leaf class contains dynamic information, its state needs to be examined and processed. Therefore, a new listener class, MyListener, needs to be added to process the event that is generated by the MyComp class. This evolution is the *packaged* pattern-level evolution defined in Section 3.1.
2. *Strategy Pattern Evolution.* In the ninth Strategy pattern instance, add a new ConcreteStrategy class, MyBufferStrategy, which is a subclass of the BufferStrategy class. This evolution is the *packaged* pattern-level evolution defined in Section 3.1.
3. *Abstract Factory Pattern Evolution.* In the Abstract Factory pattern instance, add a new ConcreteProduct, the MyPaintContext class, which is a concrete implementation of the PaintContext class. This evolution is the *correlated classes* pattern-level evolution.
4. *Adapter Pattern Evolution.* In the second Adapter pattern instance, add a new Adapter, the MyDispatcher class, which is a concrete implementation of AWTEventListener class. This evolution is the *correlated attribute/operation* pattern-level evolution.

In the following sections, we will show how to use our XSLT-based approach to help the pattern evolution processes and how to check the properties of patterns to ensure the consistency of the system design.

5.1. Evolution 1—composite pattern evolution

As introduced in Section 3.3, we developed a web-based tool, pattern evolver, to assist pattern evolution. In this section, we will show how to use this tool. First, we add all the pattern-related information to the class diagram as stereotypes defined in [9]. One of our tools, DP-Miner [18], can be used to assist this task. Then we convert the UML class diagram into an XML file by a third party tool, which will contain all descriptions of the system, including design pattern instances information. The XML file is the input of our Pattern Evolver as shown on the top panel of Figure 13. After clicking the *Process* button, all pattern instances applied in the system are listed on the left panel of Figure 13.

To perform the first evolution of the Composite pattern instance, we click on the ‘Composite[1]’ node. The node will be expanded and two possible evolutions are listed, as shown in Figure 13. One possible evolution is on Composite and the other possible evolution is on Leaf. As we are adding a new kind of component class, which plays the role of Leaf, we click the second possible evolution node and the *Add* node. After clicking *Add* node, a page to get the role information of the new pattern instance is shown on the right panel. Adding a new Leaf class and its operations does not violate the pattern properties; therefore, only one tab, *NewLeaf*, is shown. After clicking on the *NewLeaf* tab, we can input the names of the new Leaf class and its operations as shown in Figure 13. As the new component, MyComp, contains dynamic information, its state needs to be examined and processed. Therefore, a new listener class, MyListener, needs to be added to process the event that was generated by the MyComp class.

After this evolution, we use our semantic web checker to check whether the pattern properties and system design properties are violated. As shown in Figure 11, a different verification team can use our semantic web consistency checker service to import the knowledge base of this design model and the property descriptions. It checks whether the properties hold in the knowledge base and returns the result.

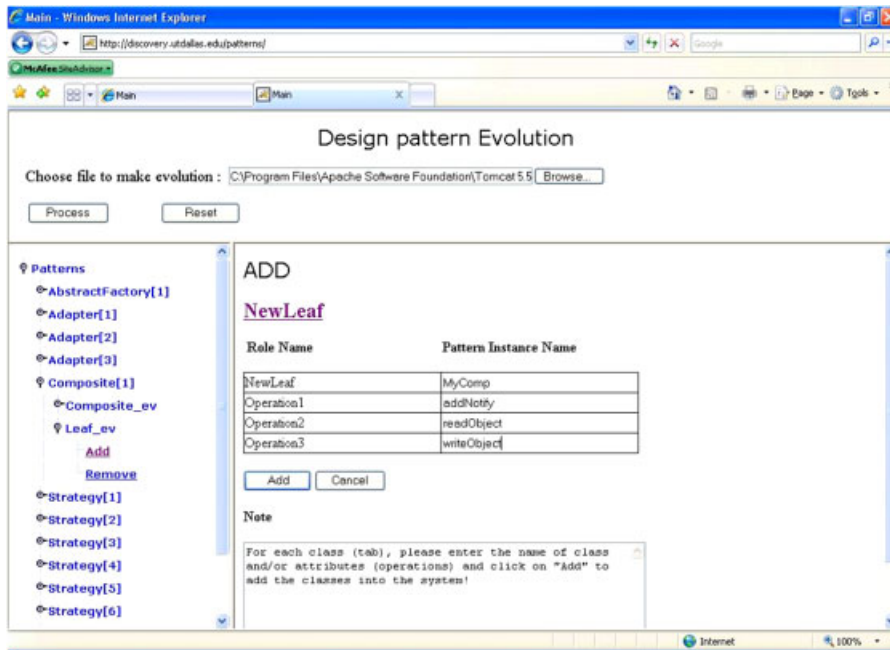


Figure 13. Pattern evolver screenshot—add new Leaf class.

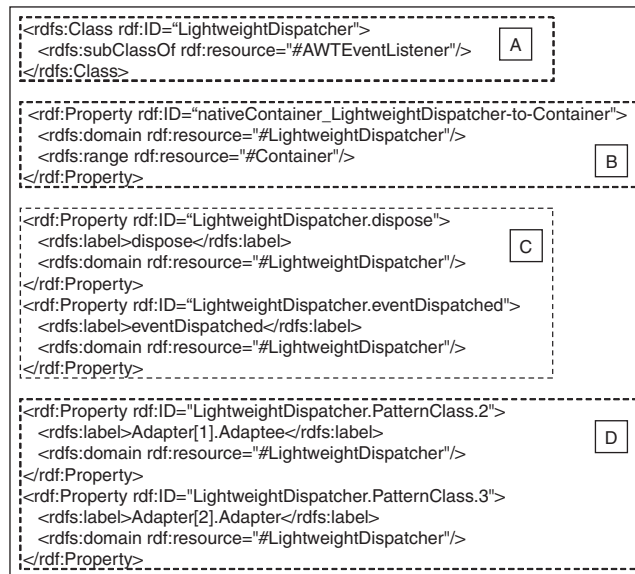


Figure 14. Partial knowledge base of the system.

The ontology of the software system design, i.e. the class model of the software system design forms the knowledge base of system on which the system consistency checking is based. For example, the RDF representation of the LightweightDispatcher class is depicted in Figure 14, where segment A represents that the name of this class is LightweightDispatcher which is a subclass of the AWTEventListener class. Segment B records the relationship of itself with other classes. The relationship in RDF is expressed as a property and the relationship ends are expressed as domain and range. Segment C states that the LightweightDispatcher class has two properties, one labeled 'dispose' and the other labeled 'eventDispatched', which are actually two operations. Segment D shows the pattern-related information of the LightweightDispatcher class. Each pattern instance in which the class involved plus its role name is expressed as a property in RDF.

Each design pattern describes expert design experience, which is characterized by some essential properties. Without the properties, the design pattern may not be considered as the pattern anymore. Thus, it is important to check whether these essential properties still hold after the pattern evolutions. The checking results may help designers to detect and correct errors in the design. In [19], we classified pattern properties into different categories and listed all properties of the GoF patterns. In the following sections, we will show the checking of the several pattern properties after each evolution. The property checking is performed by querying the design model knowledge base for certain facts. This involves the generation of queries according to the pattern properties. Currently the queries are generated manually. We will automate query generation in our future work.

Property 1.1. *Every Leaf class should be a subclass of the Component class in the Composite pattern.*

One of the properties of the Composite pattern states that the Leaf class is a subclass of the Component class. This property can be checked by two queries shown in Figure 15. The first query, (and (rdfs:label ?a 'Composite[1].Leaf') (rdfs:domain ?a ?b)), searches all the classes playing the role of Leaf in the Composite[1] instance and stores the result set in variable *b*. After the query, *b* contains the Button, Canvas, Checkbox, Choice, Label, Scrollbar, TextComponent, and MyComp classes. The second query, (and (rdfs:label ?x 'Composite[1].Component') (rdfs:domain ?x ?y) (rdfs:subClassOf ?b ?y)), searches all the classes playing the role of Component in the Composite[1] instance and their subclasses and stores the result sets in variable *y* and variable *b*, respectively. Subtracting the set stored in variable *b* of the first query from that stored in variable *b* of the second query, we should get an empty set, which means the property holds. A non-empty set means that there are Leaf classes that are not subclasses of the Component class. The checking of this property using our semantic web checker is shown in Figure 16. The message 'Successfully loaded KB' shown on the top panel indicates that the knowledge base has been successfully loaded. In the middle panel, we input the two queries together with the variables in which the results are stored,

```

First Query:
(and (http://www.w3.org/2000/01/rdf-schema#:::label ?a "Composite[1].Leaf")
      (http://www.w3.org/2000/01/rdf-schema#:::domain ?a ?b))

Second Query:
(and (http://www.w3.org/2000/01/rdf-schema#:::label ?x "Composite[1].Component")
      (http://www.w3.org/2000/01/rdf-schema#:::domain ?x ?y)
      (http://www.w3.org/2000/01/rdf-schema#:::subClassOf ?b ?y))

```

Figure 15. Query of composite pattern property 1.1.

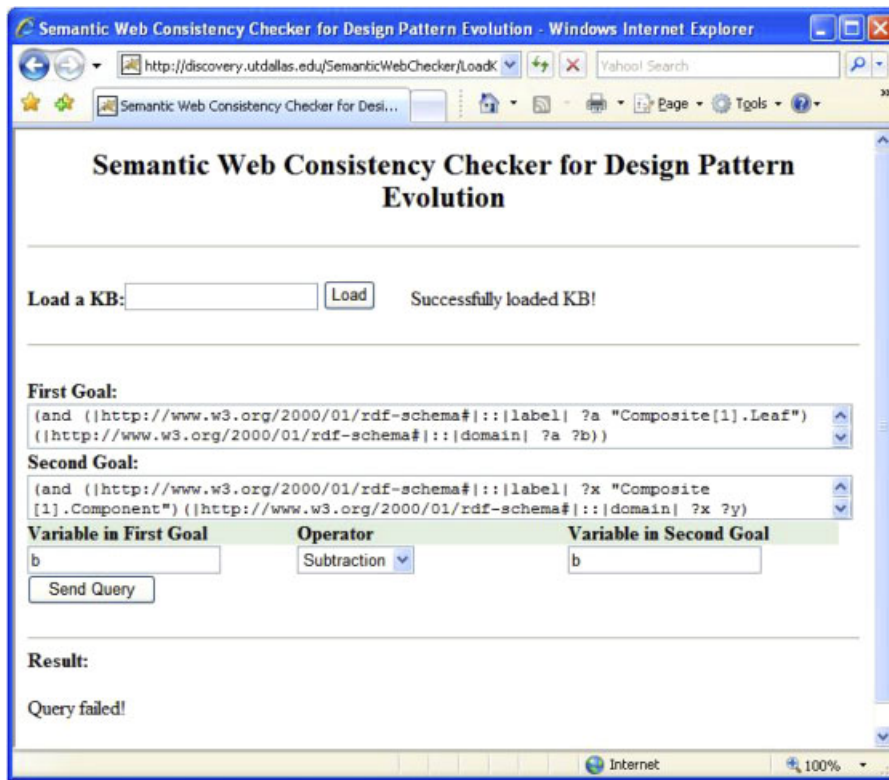


Figure 16. Screenshot of using semantic Web checker to check property 1.1.

which are both b in this case. The operation we want to perform between the two variables is subtraction, which could be selected in the drop-down list of the operators. After clicking the *Send Query* button, we get a *Query failed* result shown on the bottom panel, which means we got an empty set. Therefore, this indicates that the property holds.

Property 1.2. *Every Leaf class should have a common operation with the Component class in the Composite pattern.*

Another property specifies that the Leaf class and the Component class should have a common operation, which plays the role of Operation. This property can be described by the two queries shown in Figure 17. Variable b in the first query contains the Component class and variable y in the second query contains all the Leaf classes. $(\text{and } (\text{rdfs:label } ?t ?m) (\text{rdfs:domain } ?t ?b))$ searches all the operations in the Component class and stores the result in variable m . Similarly, the $(\text{and } (\text{rdfs:label } ?t ?m) (\text{rdfs:domain } ?t ?y))$ searches all the operations in the Leaf class and stores the result in variable m . The intersection of the value of variable m of the first query and that of variable m of the second query is a non-empty set, which means that there are common operations. The result of this query shows every Leaf class has at least one common operation with the Component class. Thus, there is no violation of this property.

```

First Query:
(and (http://www.w3.org/2000/01/rdf-schema#:::labell ?a "Composite[1].Component")
      (http://www.w3.org/2000/01/rdf-schema#:::domainl ?a ?b)
      (http://www.w3.org/2000/01/rdf-schema#:::labell ?t ?m)
      (http://www.w3.org/2000/01/rdf-schema#:::domainl ?t ?b))

Second Query:
(and (http://www.w3.org/2000/01/rdf-schema#:::labell ?x "Composite[1].Leaf")
      (http://www.w3.org/2000/01/rdf-schema#:::domainl ?x ?y)
      (http://www.w3.org/2000/01/rdf-schema#:::labell ?t ?m)
      (http://www.w3.org/2000/01/rdf-schema#:::domainl ?t ?y))

```

Figure 17. Query of composite pattern property 1.2.

```

First Query:
(and (http://www.w3.org/2000/01/rdf-schema#:::labell ?x "Strategy[1].Strategy")
      (http://www.w3.org/2000/01/rdf-schema#:::domainl ?x ?y)
      (http://www.w3.org/2000/01/rdf-schema#:::labell ?p ?o1)
      (http://www.w3.org/2000/01/rdf-schema#:::domainl ?p ?y))

Second Query:
(and (http://www.w3.org/2000/01/rdf-schema#:::labell ?x "Strategy[1].ConcreteStrategy")
      (http://www.w3.org/2000/01/rdf-schema#:::domainl ?x ?y)
      (http://www.w3.org/2000/01/rdf-schema#:::labell ?p ?o2)
      (http://www.w3.org/2000/01/rdf-schema#:::domainl ?p ?y))

```

Figure 18. Query of strategy pattern property 2.1.

5.2. Evolution 2—strategy pattern evolution

The second evolution is to add a new ConcreteStrategy class, MyBufferStrategy, which is a subclass of the BufferStrategy class in the ninth Strategy pattern instance. Similarly, we can perform this evolution using our Pattern Evolver. After adding the MyBufferStrategy class to the original design, we need to check pattern and system properties related to this pattern instance.

Property 2.1. *Every ConcreteStrategy class in the Strategy pattern instance should provide the implementation for the operation defined in the Strategy class, which plays the role of AlgorithmInterface.*

One important property of the Strategy pattern is that the ConcreteStrategy class should provide the implementation for the Strategy class. This property requires us to check whether every ConcreteStrategy class implements all the operations defined in the Strategy class, that is to say the subtraction of the operation set of the ConcreteStrategy class from that of each Strategy class should be empty. Figure 18 shows the two queries, the first of which gets the set of all the operations defined in the Strategy class in the Strategy[1] instance whose result set is stored in variable o1 and the second of which gets the set of all the operations defined in the ConcreteStrategy class in the Strategy[1] instance whose result set is stored in variable o2. If the subtraction of the result of the second query (o2) from that of the first query (o1) is an empty set, the property holds. We perform this checking on all the Strategy pattern instances, Strategies[1]–[9]. The result showed that none of them violates the property.

After examining all the existing Strategy pattern instances, we found that the newly added MyComp and MyListener class actually form the same structure with AWTEventMulticaster as

```

First Query:
(and (!http://www.w3.org/2000/01/rdf-schema#::l:label ?x "AbstractFactory[1].ConcreteProduct")
      (!http://www.w3.org/2000/01/rdf-schema#::l:domain ?x ?y))

Second Query:
(and (!http://www.w3.org/2000/01/rdf-schema#::l:label ?x "create_relation")
      (!http://www.w3.org/2000/01/rdf-schema#::l:rangel ?x ?y))

```

Figure 19. Query of abstract factory pattern property 3.1.

other components and component listeners do. It is likely to be another Strategy pattern instance. To verify our assumption, we need to check the Strategy pattern properties on these three classes. Applying the queries to the specific classes, we found that all properties are satisfied except that the AWTEventMulticaster class does not implement the myCompStateChanged operation, defined in the MyListener class, which actually violates Property 2.1. Thus, this is not an instance of the Strategy pattern. By adding the missing operation (myCompStateChanged), we have a new instance of the Strategy pattern in the design.

5.3. Evolution 3—abstract factory pattern evolution

Different from the Composite pattern and Strategy pattern examples, adding a new concrete product class into an Abstract Factory pattern instance requires not only adding the concrete product but also adding the concrete factory along with the creating methods and the creational dependency between the new concrete factory class and the new concrete product class. To prevent missing any parts of these evolutions, our Pattern Evolver provides two tabs, the NewConcreteProduct tab and ConcreteFactory tab. After providing information of the NewConcreteProduct class in the first tab, the users may click on the second tab and define the ConcreteFactory class, the MyPaint class, and the creating operation, the createContext operation. In this way, our Pattern Evolver prevents from incomplete pattern evolution and keeps the pattern properties holding throughout the system evolution.

Property 3.1. *Every concrete product in the Abstract Factory pattern shall have an associated concrete factory that creates it.*

An important property of the Abstract Factory pattern is that every concrete product shall have an associated concrete factory that creates it. This property can be expressed by the subtraction of the results of two queries, as depicted in Figure 19. The first query inquires all the classes which play the role of 'ConcreteProduct' in the AbstractFactory[1] instance with the query result stored in variable *y*. The second query searches all the classes which have the 'create' relationship and are created by other classes with the query result stored in variable *y*. The result of variable *y* in the first query returns a set containing the ColorPaintContext, GradientPaintContext, TexturePaintContext, and MyPaintContext classes. Meanwhile, the result of variable *y* in the second query returns the same set. The result of subtracting variable *y* of query 2 from variable *y* of query 1 is an empty set. After conducting this query for each Abstract Factory pattern instance, in this case there is only one, we can conclude that every concrete product has an associated concrete factory in the evolved system design. At this point, we can see how our pattern evolver tool helps preventing inconsistency in system evolution.

5.4. Evolution 4—adapter pattern evolution

Adding a new dispatcher class playing the role of Adapter class in an Adapter pattern instance is similar to the previous evolution, which requires adding more than one class. This evolution requires not only adding the NewAdapter class and defining its operations, but also defining the Adaptee class and the related operations. Therefore, our Pattern Evolver tool will display two tabs, NewAdapter and Adaptee. The second tab reminds the user to add information about the associated Adaptee class, as well as the new Adapter class. The associated Adaptee class does not have to be a new class. It can be an existing class, such as the Container class, and the `getMouseEventTarget` operation that was originally defined in the Container class can serve as the `SpecificRequestion` operation.

Property 4.1. *Every Adapter class in the Adapter pattern instance should have an association relationship with an Adaptee class.*

There are some properties related to the Adapter pattern. As an example, we check the property that every Adapter class in every Adapter pattern instance is associated with an Adaptee class (Property 4.1). This property can be expressed by the subtraction of the result sets of two queries, as shown in Figure 20. The first query searches all the subclasses of the Target class in the Adapter[1] instance and stores the results in variable *c*, because the subclasses of the Target class are the Adapter classes. The second query searches all the classes that have association relationships with the Adaptee class in the Adapter[1] instance and stores the results in variable *c*. Subtracting these two sets, we should end up with an empty set, which indicates that there is no violation of this property. If the result of subtraction is not empty, it means that there are Adapter classes that do not have any association relationship with the Adaptee class. We perform this checking for Adapter[1] through Adapter[3]. The results of Adapter[1] and Adapter[2] are empty sets, whereas that of Adapter[3] is not empty, which includes the MyBufferStrategy class. We found that an association relationship between the MyBufferStrategy class and the BufferCapabilities class is missing. Thus, Property 4.1 does not hold in the third instance of the Adapter pattern. Examining the design in Figure 12, we found that the main reason for this problem is caused by the addition of a new ConcreteStrategy class in Section 5.2. The ConcreteStrategy classes participate in two patterns, Strategy and Adapter. Although our Pattern Evolver may avoid the inconsistent evolution in the Strategy pattern, it does not prevent the interaction with other patterns. Thus, it is important to check the properties of the patterns as described in this section. After modifying the evolved system by adding the association relationship between the MyBufferStrategy class and the BufferCapabilities class, our checker shows that Property 4.1 holds this time.

5.5. Revised design

Based on the inconsistencies that we found using our semantic web checker, we change our design to satisfy the desired system design properties and pattern properties. Figure 21 shows the system design after changing. Comparing with Figure 12, the AWTEventMulticaster in Figure 21 contains one more operation, `myCompStateChanged`, which was added after noticing the violation of Property 2.1. The new diagram also contains one more association relationship between the MyBufferStrategy and the BufferCapabilities class, which was added after noticing the violation of Property 4.1. Further, it contains one more class, the MyPaint class, and one more create

```

First Query:
(and (http://www.w3.org/2000/01/rdf-schema#:::label ?x "Adapter[1].Target")
      (http://www.w3.org/2000/01/rdf-schema#:::ldomain ?x ?y)
      (http://www.w3.org/2000/01/rdf-schema#:::lsubClassOf ?c ?y))

Second Query:
(and (http://www.w3.org/2000/01/rdf-schema#:::label ?x "Adapter[1].Adaptee")
      (http://www.w3.org/2000/01/rdf-schema#:::ldomain ?x ?y)
      (http://www.w3.org/2000/01/rdf-schema#:::lrangel ?t ?y)
      (http://www.w3.org/2000/01/rdf-schema#:::ldomain ?t ?c))

```

Figure 20. Query of adapter pattern property 4.1.

relationship between the MyPaint and MyPaintContext classes, which was added during the evolution with the help of our pattern evolver tool.

5.6. Summary

This case study has shown that our Pattern Evolver greatly improves the process of system evolution. It helps the system understanding by listing all pattern instances discovered during reverse engineering process as well as their participants. It automates the evolution process by gathering evolution information from engineers and updating the design automatically. It ensures the integrity of the design pattern instances during the evolution by restricting the possible evolutions, i.e. only the evolutions defined in XML are allowed and the evolutions not following the rules will be prevented.

This case study also shows the necessity of using Semantic Web Checker to detect design mistakes. Maintaining the system design information, the consistency checking is able to answer the queries related to system design. Especially, we want to ensure that inconsistencies are not introduced during the evolution process. With the queries about the design pattern properties, therefore, the inconsistencies among pattern instances can be detected. Any error that is not prevented during the evolution process by our Pattern Evolver will be identified in this validation phase.

6. RELATED WORK

There are several research works on design pattern evolutions. The XML and XSLT techniques have also been applied in many different applications. To the best of our knowledge, however, model transformations based on XSLT for design pattern evolutions have not been investigated. Model transformations naturally integrate with the MDA making the applications working with large-scale real-life systems.

Design pattern evolutions in software development processes are discussed in [20], where software development processes are considered as the evolutions 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. Our approach also automates the evolutions based on model transformations by XSLT.

Noda and Kiski [21] consider design patterns as a concern that is separated from the application core concern. Thus, an application class may assume a role in a design pattern by weaving the

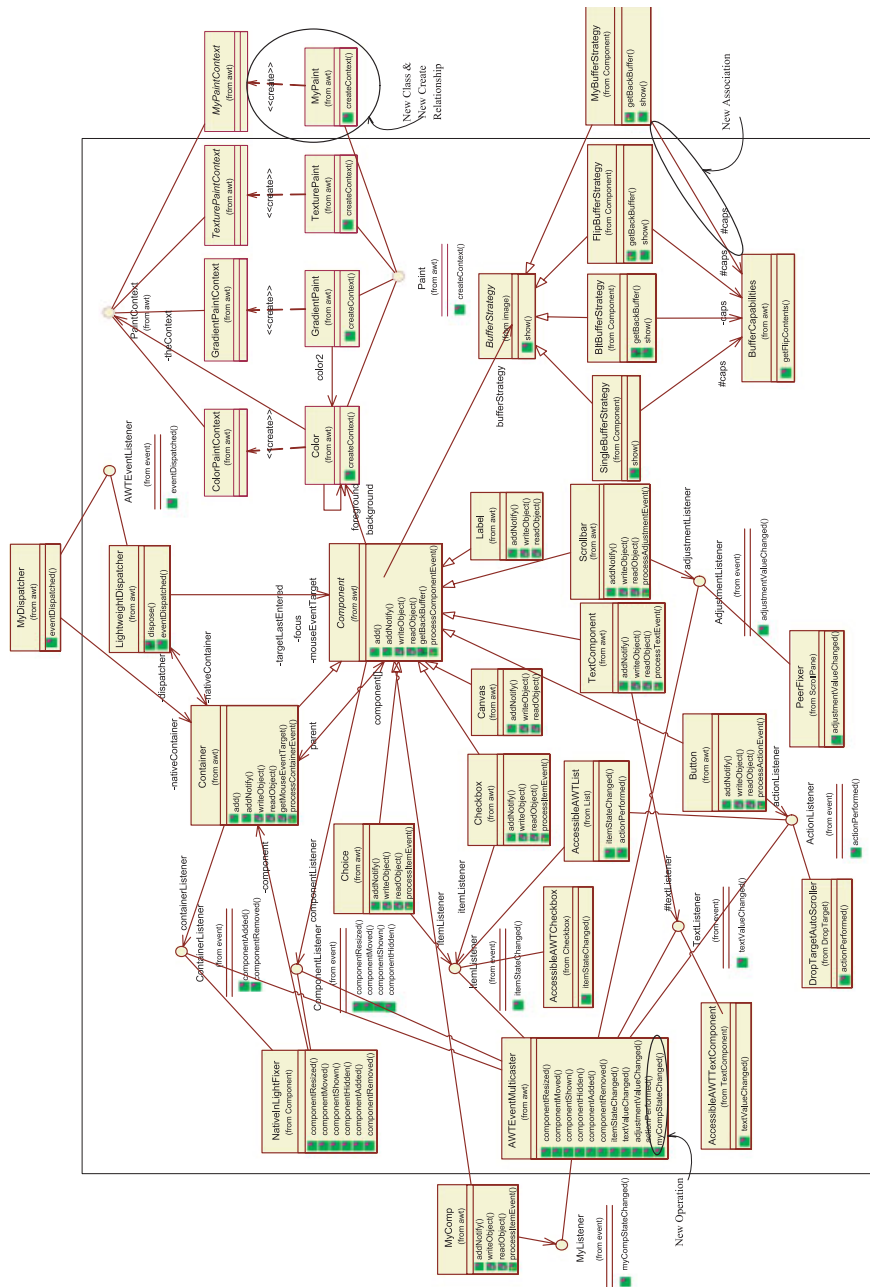


Figure 21. Partial AWT design and its evolutions—properties checked.

design pattern concern into the application class using Hyper/J [22]. Owing to the separation of concerns, an application class may assume different roles in different design patterns. The change of the roles that an application class plays, i.e. the change of design patterns, becomes a relative simple task. The main goal of their evolution of design pattern is the replacement of one pattern with 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.

Aoyama [23] proposed a set of concepts, representations, and methodology to understand design evolution in terms of design pattern. An intuitive understanding of both design patterns and design evolution is provided. They summarize the two types of design evolution, intensive evolution and extensive evolution, and represent design evolution as relationships among design patterns using pattern evolution diagrams. Differently, we treat design pattern evolutions as XMI transformation. Using XSL transformation rules, we could transform the original system specification in XMI to an evolved version of system in XMI. After the transformation, the system could be changed into the RDF format and checked for system consistency.

Improving software system quality by applying design patterns in existing systems has been discussed in [24]. 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 minipatterns. The main goal of their software evolution is to apply design patterns in existing systems, whereas our evolution goal is to change the patterns that have already applied in a system.

Aversano *et al.* [25] present the results from an empirical study aimed at the evolution of design patterns in large software systems. The results indicate that the pattern changed frequently and the amount of co-change does not depend on the pattern type, but rather on the role played by the pattern to support the application features. Patterns are often changed either in their implementation or by adding subclasses or changing method interfaces: the latter however causes a higher co-change on client classes. Their results suggest that developers should carefully consider pattern usage when it supports crucial features of the application. Different from their study, our research focuses on automating the process of design pattern evolution and the process of consistency checking after the evolution.

The evolution processes of design patterns have been studied in [26], where Prolog [27] is used to capture the structural evolution processes of design patterns. The structural aspect of a design pattern is described in terms of Prolog facts. Thus, the evolution of a design pattern application can be achieved by the addition or removal of new or old Prolog facts. The evolution processes are defined as Prolog rules. In this paper, we describe the evolution process as model transformations based on XMI and XSLT in terms of two-level evolutions [8,28]. Different from Prolog rules, model transformation rules can be automated in the MDA. Consistency checking is also provided to ensure the properties of each pattern. Model transformations can be naturally integrated to existing object-oriented technology to work with larger-scale applications, whereas Prolog rules are typically hard to scale up.

A graph transformation-based approach that is able to evolve design patterns and extend the system is presented in [29]. Design patterns expressed in UML class diagrams are extracted and depicted in the graph editor. The graph transformation engine consists of a set of predefined graph transformation rules for each type of pattern evolution. It takes the UML diagrams as input, accepts user-desired modifications as commands, and generates the evolved design pattern and output. To verify the consistency after transformation, a grammar parser is proposed and productions

are defined for each design pattern. Instead of using graph transformation rules, we used XSLT transformation rules that are applied on XMI formatted system specification to perform the pattern evolution. Instead of grammar parser, we use semantic web checker and RDF to conduct the system consistency checking. Thus, it allows us to perform pattern evolutions and consistency checking in larger systems.

Kim and Chang [30] integrated service-oriented design and the concept of aspects in the design phase of software life cycle to utilize services and aspects as fundamental and abstract elements. Lu [31] applied SOA to the distributed web GIS application and integrated web services, Servlet/JSP functions and GIS APIs in a multi-layer architecture. In contrast to these approaches, we define an SOA for the evolutions and analyses of design patterns at different locations based on our initial work [14].

A generic XMI-based transformation infrastructure of UML models has been presented in [32]. This allows the user to select a predefined generic XML-based transformation and configure its parameters. Unlike this work, we concentrate on the XMI-based transformations for design pattern evolution.

The tool support for UML model evolution is provided in [33], which is also based on XMI. The design and development of the tool applies several design patterns. In contrast, we focus on the evolution of design patterns, instead of the evolution of UML models.

Experiments have been conducted in [34] to show that XMI can be used to transform the UML models into other modeling languages, such as SQL. The implementation of the XMI-based transformation uses XSLT. We base on these experiments and use XSLT to implement the transformations.

Kalnins *et al.* [35,36] 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. Muller *et al.* [37] 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. Thus, it is better to use some low-level transformation language such as XSLT to achieve the goal.

7. CONCLUSIONS

As 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 characterize two-level transformations: the primitive level and the pattern level, and explicitly describe design pattern evolutions using these two-level transformations. We also implement the transformation based on XMI using a XSLT processor. The evolutions of each design pattern are defined as XSLT transformation rules. In addition, our semantic web checker can check the constraints of evolutions of each design pattern after evolutions.

In the future, we will design a more rigorous RDF vocabulary to define software systems, including design pattern information, so that we will have less ambiguity when checking for system or design properties. We will also try to apply semantic web techniques for software system modeling, design pattern visualization, etc.

REFERENCES

1. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley: Reading, MA, 1995.
2. Model Driven Architecture. <http://www.omg.org/mda/> [December 2008].
3. Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide*. Addison-Wesley: Reading, MA, 1999.
4. W3C. Extensible Markup Language (XML). <http://www.w3.org/> [December 2008].
5. W3C. XSL Transformations (XSLT). <http://www.w3.org/> [December 2008].
6. Fikes R, Jenkins J, Frank G. JTP: A system architecture and component library for hybrid reasoning. *Proceedings of the Seventh World Multiconference on Systemics, Cybernetics, and Informatics*, Orlando, FL, U.S.A., July 2003.
7. JTP. <http://www.ksl.stanford.edu/software/JTP/> [December 2008].
8. Dong J, Yang S, Zhang K. A model transformation approach for design pattern evolutions. *Proceedings of the Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS)*, Germany, March 2006; 80–89.
9. Dong J, Yang S, Zhang K. Visualizing design patterns in their applications and compositions. *IEEE Transactions on Software Engineering (TSE)* 2007; **33**(7):433–453.
10. Rational Rose website. <http://www.rational.com/> [September 2006].
11. ArgoUML. <http://argouml.tigris.org/> [December 2008].
12. MetaData Repository. <http://mdr.netbeans.org/> [December 2008].
13. IBM. http://www-128.ibm.com/developerworks/rational/library/05/503_sebas/ [September 2006].
14. Dong J, Yang S, Lad DS, Sun Y. Service oriented evolutions and analyses of design patterns. *Proceedings of the Second IEEE International Symposium on Service-oriented System Engineering*, Shanghai, China, October 2006; 11–18.
15. RDF. <http://www.w3.org/RDF> [December 2008].
16. XMIR2RDF. <http://freshmeat.net/projects/ju2d/> [September 2006].
17. Java.awt resource information. <http://java.sun.com/j2se/1.5.0/docs/guide/awt/index.html> [September 2006].
18. Dong J, Lad DS, Zhao Y. DP-Miner: Design pattern discovery using matrix. *Proceedings of the Fourteenth Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS)*, Arizona, U.S.A., March 2007; 371–380.
19. Dong J, Zhao Y. Classification of design pattern traits. *Proceedings of the Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Boston, U.S.A., July 2007; 473–476.
20. Kobayashi T, Saeki M. Software development based on software pattern evolution. *Proceedings of the Sixth Asia-Pacific Software Engineering Conference (APSEC)*, Takamatsu, Japan, 1999; 18–25.
21. Noda N, Kishi T. Design pattern concerns for software evolution. *Proceedings of the 4th International Workshop on Principles of Software Evolution*, Vienna, Austria, 2001; 158–161.
22. Hyper/J. <http://www.alphaworks.ibm.com/tech/hyperj>.
23. Aoyama M. Evolutionary patterns of design and design patterns. *Proceedings of International Symposium on Principles of Software Evolution*, Kanazawa, Japan, 2000; 110–116.
24. Ó Cinnéide M, Nixon P. Automated software evolution towards design patterns. *Proceedings of the International Workshop on the Principles of Software Evolution*, Vienna, Austria, September 2001; 162–165.
25. Aversano L, Canfora G, Cerulo L. An empirical study on the evolution of design patterns. *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, 2007; 385–394.
26. Alencar P, Cowan D, Dong J, Lucena C. A pattern-based approach to structural design composition. *Proceedings of the IEEE 23rd Annual International Computer Software and Applications Conference*, Phoenix, U.S.A., October 1999; 160–165.
27. Clocksin WF, Mellish CS. *Programming in Prolog*. Springer: Berlin, 1987.
28. Dong J, Yang S, Huynh DT. Evolving design patterns based on model transformation. *Proceedings of the Ninth IASTED International Conference on Software Engineering and Applications (SEA)*, U.S.A., November 2005; 344–350.
29. Zhao C, Kong J, Dong J, Zhang K. Pattern based design evolution using graph transformation. *International Journal of Visual Languages and Computing (JVLC)* 2007; **18**(4):378–398.
30. Kim T, Chang C. Service-oriented design with aspects (SODA). *Proceedings of the 2005 IEEE International Conference on Services Computing (SCC'05)*, Florida, U.S.A., July 2005; 319–322.
31. Lu X. An investigation on service oriented architecture for constructing distributed Web GIS application. *Proceedings of the 2005 IEEE International Conference on Services Computing (SCC'05)*, Orlando, FL, U.S.A., July 2005; 191–197.

32. Kovse J, Harder T. Generic XMI-based UML model transformations. *Proceedings of the International Conference on Object-oriented Information Systems*, Montpellier, September 2002. Springer: Berlin, 2002; 192–198.
33. Keienburg F, Rausch A. Using XML/XMI for tool supported evolution of UML models. *Proceeding of International Conference Hawaii International Conference on System Science*, Maui, HI, January 2001; 3945–3954.
34. Demoth B, Hussmann H, Obermaier S. Experiments with XML-based transformations of software models. *Workshop on Transformations in UML (ETAPS 2001 Satellite Event)*, Genova, April 2001.
35. Kalnins A, Barzdins J, Celms E. Model transformation language MOLA. *Proceedings of MDAFA 2004 (Model-driven Architecture: Foundations and Applications 2004)*, Linköping, Sweden, June 2004; 14–28.
36. Kalnins A, Barzdins J, Celms E. Model transformation language MOLA: Extended patterns. *Sixth International Baltic Conference DB@IS 2004*, vol. 118. IOS Press: FAIA, 2005; 169–184.
37. Muller P-A, Fleurey F, Vojtisek D, Drey Z, Pollet D, Fondement F, Studer P, Jezequel JM. On executable Meta.Languages applied to model transformations. *Proceedings of INRIA Workshop of Model Transformations in Practice*, Half Moon Resort, Montego Bay, Jamaica, October 2005.