

Service Oriented Evolutions and Analyses of Design Patterns

Jing Dong, Sheng Yang, Dushyant S. Lad

*Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083, USA
{jdong, syang, dsl044000}@utdallas.edu*

Yongtao Sun

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

Abstract

The globalization of software development helps to reduce business cost by outsourcing software design and development tasks. However, it also poses new challenges on the collaboration and management of the development teams at multiple remote sites. Service-oriented approaches facilitate web-based collaborations and reuses. In this paper, we propose a service-oriented architecture for the collaboration of software design teams in distributed locations. In particular, our approach allows different design teams from multiple remote sites to design software systems with design patterns, to evolve the designs, and to check the consistencies of the design evolutions. We provide service-oriented prototype tools for the design teams from different physical locations to model the software system designs in UML, to evolve the designs based on XSLT transformations, and to check the consistencies based on the Java Theorem Prover. A case study is presented to illustrate our approach.

1. Introduction

Software development becomes more and more a global process that involves project teams from all over the world working in concert to achieve a common goal. Due to the economic pressure on cutting business cost, outsourcing software development tasks from developed countries to developing countries becomes a common practice in software industry. Even medium and small companies start to open business branches in different countries. As a result, the development of a single software product may involve project teams from different countries. Thus, the new challenge is the collaboration and management of the development teams at multiple remote sites which work together for the same project. In particular, this kind of collaboration may even be within a particular software development phase, such as design and testing. For example, a software system may be designed by a team from one location, analyzed by a team from another location, and changed by a team from the third location.

Service-oriented computing (SOC) in system engineering [9] 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 SOC, 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 vocabulary and structure of messages. An extensible schema allows new versions of services to be introduced without breaking existing services.

To facilitate the on-line collaborations among design teams from different physical locations, we propose a service-oriented architecture for the collaboration and management of software designs, evolutions, and analyses from multiple remote sites. In our approach, teams from different locations may be responsible of different design tasks. For example, some team may design a software system using design patterns as a design language. The resulting design in UML [1] may be changed by another team from a different location. This evolution may cause some inconsistencies in the system designs which can be checked by yet another team from a different location. We provide service-oriented prototype tools that allow the design teams to design a software system, to evolve the design, and to

analyze the design evolution from multiple remote sites. In this paper, we approach the collaborative design and evolution on design patterns [4] which document expert design practice. We provide services that can change the pattern-based design by adding or removing design elements in existing design pattern applications. Although we present our service-oriented architecture in terms of pattern-based design, our approach is applicable to general designs by defining possible changes of a design as evolution processes and analyzing the change impact using our services. To illustrate our approach, we present a case study that includes a design with several design patterns, their evolutions, and analyses of several properties.

The remainder of this paper is organized as follows: the next section describes the service-oriented architecture including web-based pattern evolver and semantic web consistency checker. Section 3 presents a case study to show our approach on the on-line design collaborations. The last two sections cover related work and conclusions.

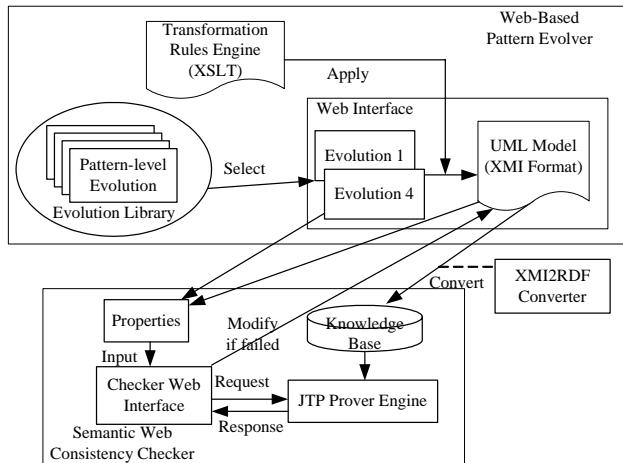


Figure 1 The Overall Architecture of the Approach

2. Service-Oriented Architecture

At software design phase, design patterns [4] have become the design language for software teams to communicate design decisions and models. One of the important goals of design patterns is design for change. Thus, most of the design patterns encapsulate future changes that may only affect limited part of a design pattern. In the document of each design pattern, however, the evolution information is generally not explicitly specified. When changes are needed, a designer has to read between the lines of the document of a design pattern to figure out the correct ways of changing the design. Misunderstanding of a design pattern may result in missing parts of the evolution process. The addition and removal of system parts should not violate the constraints and properties of

design patterns. Thus, it is important to have, in the documentation of the design pattern, information about the evolution of the patterns. The evolution of a software system at the design level is less costly than it is at the implementation level.

Figure 1 depicts the service-oriented architecture of our approach that consists of two parts: the web-based pattern evolver and semantic web consistency checker. The former allows the developers to change the pattern-based design at different sites. The latter can check the consistencies of evolutions based on the semantic web.

2.1 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 [2][13].

Evolution library. A design pattern normally encapsulates some particular ways for future changes. However, such evolution information of each design pattern is normally implicit to the description of the pattern. A designer has to search the document of the pattern to find the guidance on evolution. Misunderstanding and mistakes of changing the design patterns may compromise the benefits of using these patterns and have huge impact on the system designs. To capture the evolution processes of design patterns, we define two-level evolutions: the primitive level and the pattern level in [2]. The primitive-level evolutions are the addition or removal of modeling elements, such as classes and relationships. We identify nine modeling elements that can be added or deleted as the primitive-level evolutions [2]. The pattern-level evolution characterizes the recurring evolution processes which occur in many design patterns. It is described in terms of a sequence of the basic primitive-level evolutions. Each design pattern may perform some of the pattern-level evolutions, which can be added in the document of the pattern. We characterize five pattern-level evolutions that are recurring in different design patterns [2]. We studied the types of pattern evolutions of the design patterns listed in [4] and build an evolution library.

Based on our previous work on categorizing pattern evolutions [2], we define the possible evolutions of each design pattern in an XML file as a service library in this paper. This service library provides the candidate evolutions for our pattern evolver service. For instance, the Adapter pattern has two possible pattern-level evolutions: add a new adapter into the pattern or add a new adaptee into the pattern [2]. Figure 2 shows one of the possible evolutions of the Adapter pattern, adding a new adaptee. For this evolution, a class which plays a role Adaptee in the Adapter pattern is added into the

design pattern. Meanwhile, a class which plays a role of Adapter in the Adapter pattern is added simultaneously. The Adaptee class has an operation SpecificRequest defined and the Adapter class has an operation Request.

```
<?xml version="1.0" encoding="UTF-8"?>
<patterns xmlns="http://www.patterns.edu"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.patterns.edu patterns.xsd">
  <pattern>
    <pattern_name>Adapter</pattern_name>
    <evolution>
      <evolution_name>Adaptee</evolution_name>
      <class>
        <class_name>NewAdaptee</class_name>
        <operation>
          <operation_name>SpecificRequest</operation_name>
        </operation>
      </class>
      <class>
        <class_name>NewAdapter</class_name>
        <operation>
          <operation_name>Request</operation_name>
        </operation>
      </class>
    </evolution>
    <evolution>
      ...
    </evolution>
  </pattern>
</patterns>
```

Figure 2 Pattern-Level Evolution Definition in XML

Transformation rule engine. In order to automatically evolve design patterns, the evolution processes are implemented based on XMI format to transform the UML models of design pattern applications. Thus, the structure of a design pattern may evolve in some prescribed ways based on the pattern-level transformation. The evolution processes of design patterns are specified as XSLT transformation rules based on the two-level evolutions. These transformation rules define which modeling elements are changed (added or removed) and 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 XMI format. Thus, the pattern-level evolution is a sequence 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 model elements corresponding to this pattern-level evolution into/from the original UML model in XMI format.

Web interface. The selection of design pattern evolutions and the invocation of the system evolution can be done through web interface shown in Figure 3 and Figure 4. Figure 3 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 and 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 in the left panel of web interface in Figure 3. 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 applied to the system design.

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

When the user wants to remove model element(s) from the system, he can select “remove” item from the pattern tree in left panel of the user interface. Figure 4 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 dropdown list in the right panel of the user interface. The user can select the class he wants to remove from the Adapter pattern and click on the “Remove” button to remove the select class from the system.

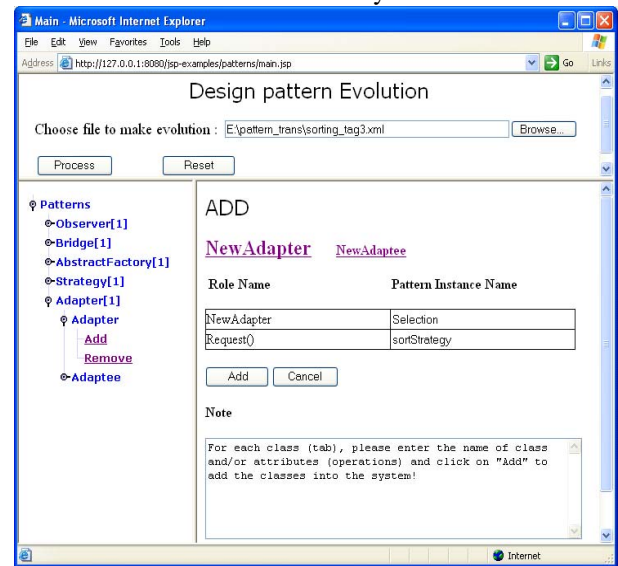


Figure 3 Pattern Evolution Automation System – Addition

UML model (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 [14] or ArgoUML [11]. Since 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 XMI format.

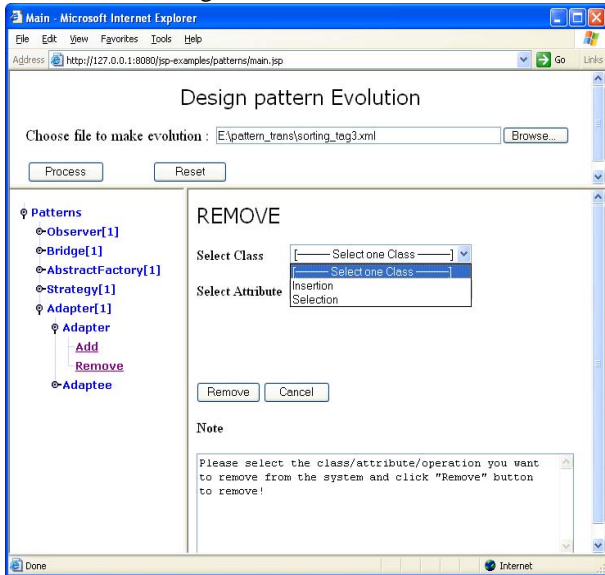


Figure 4 Pattern Evolution Automation System – Removal

2.2 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. Thus, we need to check that the evolved system has the same properties as the original one. In order to assure the essential properties of a design pattern still hold after the evolutions, we provide a consistency checker service based on the Java Theorem Prover (JTP) [3][15] 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 based reasoning system.

Knowledge base. In order to check the consistency of the evolved design, the evolved design is first converted into RDF [16] format from the XMI format. Thus, the consistency checker can be used to perform the system consistency checking. This conversion can be partially automated by the XMI2RDF converter [17].

In RDF/RDFS, a class is defined in `<rdfs:Class>` tag with `rdf:ID` as its attribute. The value of the attribute of `<rdfs:Class>` defines the class name. The generalization relationship is defined in `<rdfs:subClassOf>` tag. The association relationship is considered as a property of the class. For instance, Figure 5 shows a partial knowledge base of the Observer pattern in RDF/RDFS format. Lines 1 and 2 define classes Subject and Observer, respectively. Lines

3 to 5 define class ConcreteSubject, which is a subclass of Subject. Lines 6 through 9 define an association between classes Subject and Observer.

```

1 <rdfs:Class rdf:ID="Subject"/>
2 <rdfs:Class rdf:ID="Observer"/>
3 <rdfs:Class rdf:ID="ConcreteSubject">
4   <rdfs:subClassOf rdf:resource="#Subject"/>
5 </rdfs:Class>
6 <rdf:Property rdf:ID="{Subject-Observer}_Subject-to-Observer">
7   <rdfs:domain rdf:resource="#Subject"/>
8   <rdfs:range rdf:resource="#Observer"/>
9 </rdf:Property>

```

Figure 5 Representing Design Pattern in RDF/RDFS

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 Concrete Factory class should also be added into the pattern application. Otherwise the properties of the Abstract Factory pattern no longer hold. Our approach can avoid this kind of problem 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 this kind of inconsistencies. Thus, we need to perform system inconsistency check 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, which are represented by a triple. 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 6 shows an example of the query in JTP, which tests whether there

is circular inheritance in the design, i.e., the class ConcreteSubject is a subclass of the class Subject which is also a subclass of the class ConcreteSubject.

```
(and ([http://www.w3.org/2000/01/rdf-schema#::subClassOf] |file:/F:/syang/JTP/observer.xml#::ConcreteSubject| |file:/F:/syang/JTP/observer.xml#::Subject|)
([http://www.w3.org/2000/01/rdf-schema#::subClassOf] |file:/F:/syang/JTP/observer.xml#::Subject| |file:/F:/syang/JTP/observer.xml#::ConcreteSubject|))
```

Figure 6 A Query in RDF/RDFS

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 RDF format. The outputs are matching results found by JTP prover based on the loaded knowledge base.

Checker web interface. At the client site, we provide a web interface for the semantic web consistency checker as shown in Figure 7. It facilitates the consistency checking by a team from a remote site. There are three main parts of the web interface: load KB, queries, and checking result. First, the knowledge base of the system design needs to be loaded into the

checker. As described previously, the knowledge base can be generated from the UML design model. In the second part, we allow the user to conduct two queries for two different goals (first goal and second goal). The main reason is 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 shows that the property holds, the user may continue checking other properties. If the result shows that the property does not hold, the user may analyze the design model based on the checking result to find the problem. After correcting the problem, the user may check the same property against the modified knowledge base.

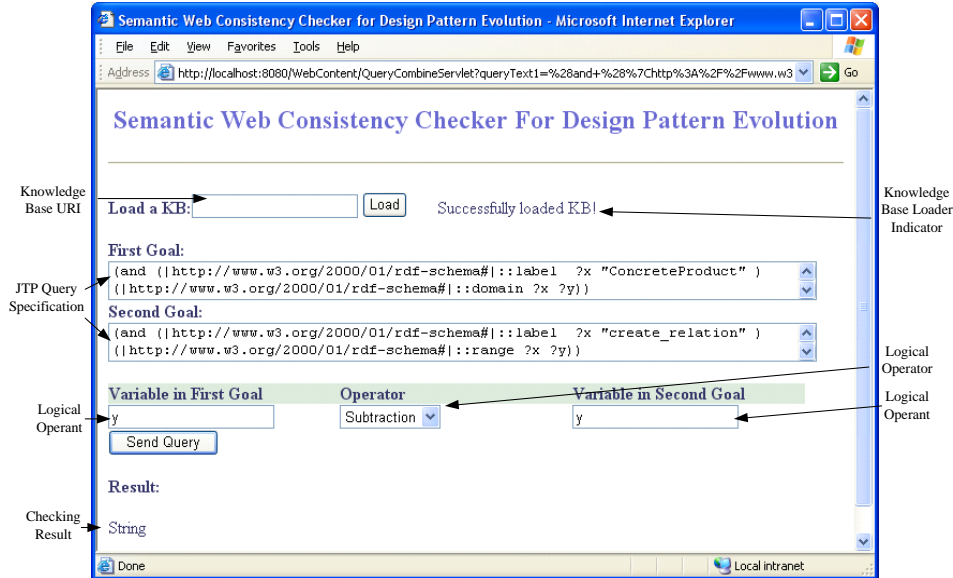


Figure 7 System Consistency Check Web Interface

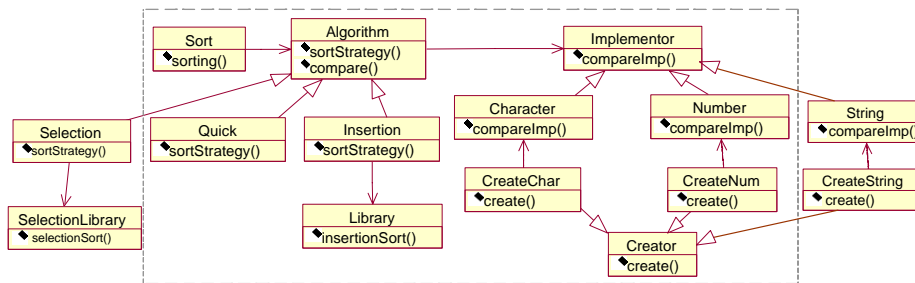


Figure 8 Sorting System Design and its Evolutions

3. Case Study

In this section, we present a case study related to a sort example to illustrate our service-oriented approach for collaborative design. The UML model in the dashed box of Figure 8 depicts the design that can sort different types (e.g. Character and Number) of elements using different sort algorithms (e.g. Quick sort and Insertion sort). The design of the sort example contains four design patterns: the Strategy, Adapter, Bridge, and Abstract Factory patterns. To dynamically change the sort algorithms, the Strategy pattern consisting of the Sort, Algorithm, Quick, Insertion, and Library classes is applied. Suppose there is a library function for the insertion sort with a different interface. The Adapter pattern, including the Algorithm, Insertion, and Library classes, is applied to resolve the interface difference. The Bridge pattern is applied to facilitate the sorting of different types of elements, which includes the Algorithm and Implementor classes and their children classes. The Abstract Factory pattern is applied to create families of products, which includes the Implementor and Creator classes and all their children classes.

Let us consider a scenario in the sort design shown inside the dashed box in Figure 8. The sort system designed by one team in a location initially has two sort algorithms available, i.e., the Quick and Insertion sort. Later on, suppose another sort algorithm (selection sort) is available in a new library. However, the interface provided by the library is different from the one the user needs. The Adapter pattern can evolve by another design team from a different location with the addition of the Selection and SelectionLibrary classes (left hand side of Figure 8) in the system design. The Selection class plays the role of Adapter, whereas the SelectionLibrary class plays the role of Adaptee in the Adapter pattern. As shown in our service in Figure 3, therefore, there are two hyperlinks: one for the class which plays the role of Adapter and the other for the class which plays the role of Adaptee in the right panel of the user interface. The user can enter the name of the new Adapter class (Selection in Figure 3) in the right column of the table. The operation `sortStrategy()` is the required interface in the system design and plays the role of Request in the Adapter pattern. Similarly, the user can enter the new Adaptee class (SelectionLibrary). By clicking “Add” button, the new model elements can be automatically added into the original system based on model transformations.

Figure 4 shows the interface through which the user can delete one of the Adapter classes (Selection) from the sort system design. When the user click on the Remove item from the design pattern tree in the left panel of the interface, all classes which play a role of Adapter are shown in a drop-down box. The user can select the class

which needs to be removed from the system (e.g. Selection in Figure 4) and by invoking the remove action, the class Selection can be removed. Consequently, the associated Adaptee class, SelectionLibrary, is also removed from the system design simultaneously.

As discussed previously, we need to perform consistency checking to ensure the evolved software system still has the proper properties after the evolution. As shown in Figure 7, 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.

The ontology of the software system design, i.e., the class diagram 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 Selection class is depicted in Figure 9, where the segment A represents that the Selection class is a subclass of the Algorithm class. Segment B records the relationship between classes Selection and SelectionLibrary. The relationship is expressed in RDF as a property and the relationship ends are expressed as domain and range in RDF. Segment C states that the Selection class has a property with label “sortStrategy”. Segment D shows the pattern-related information of the Selection class. Each pattern name, role name and instance is expressed as a property in RDF. Let us consider the following properties.

Property 1: *the Adapter classes and the Target class shall have the same operation interface in the Adapter pattern.*

This property can be proved by the subtraction of the results of the two separate JTP queries. The first query searches all the classes which play the role of Adapter and Target in the Adapter pattern as shown in Figure 10. The result of the query is recorded in variable ?y which contains three classes: Insertion, Selection, and Algorithm. The second query finds all classes having the “sortStrategy” operation as shown in Figure 11. The query result, stored in variable ?y, consists of four classes: Insertion, Selection, Quick and Algorithm. These two queries can be inputted in the First goal and Second goal, respectively, in the web interface shown in Figure 7. The subtraction of the result of variable ?y from query one and that of ?y from query two is an empty set, which means that all classes play the roles of Target and Adapter define the “softStrategy” operation (the same interface). Thus, we can conclude that all classes which have a role Adapter or Target in the Adapter pattern have “sortStrategy” interface defined, which proves the Property 1.

```

<rdf:Class rdf:ID="Selection">
  <rdf:subClassOf rdf:resource="#Algorithm"/>
</rdf:Class>
<rdf:Property rdf:ID="Selection-SelectionLibrary_Selection-to-SelectionLibrary">
  <rdf:domain rdf:resource="#Selection"/>
  <rdf:range rdf:resource="#SelectionLibrary"/>
</rdf:Property>
<rdf:Property rdf:ID="Selection.sortStrategy">
  <rdf:label> sortStrategy </rdf:label>
  <rdf:domain rdf:resource="#Selection"/>
  <rdf:range rdf:resource="#integer"/>
</rdf:Property>
<rdf:Property rdf:ID="Selection.PatternClass.pattern.1">
  <rdf:label> Strategy </rdf:label>
  <rdf:domain rdf:resource="#Selection"/>
</rdf:Property>
<rdf:Property rdf:ID="Selection.PatternClass.role.1">
  <rdf:label> ConcreteStrategy </rdf:label>
  <rdf:domain rdf:resource="#Selection"/>
</rdf:Property>
<rdf:Property rdf:ID="Selection.PatternClass.instance.1">
  <rdf:label> 1 </rdf:label>
  <rdf:domain rdf:resource="#Selection"/>
</rdf:Property>
<rdf:Property rdf:ID="Selection.PatternClass.pattern.2">
  <rdf:label> Adapter </rdf:label>
  <rdf:domain rdf:resource="#Selection"/>
</rdf:Property>
<rdf:Property rdf:ID="Selection.PatternClass.role.2">
  <rdf:label> Adapter </rdf:label>
  <rdf:domain rdf:resource="#Selection"/>
</rdf:Property>
<rdf:Property rdf:ID="Selection.PatternClass.instance.2">
  <rdf:label> 1 </rdf:label>
  <rdf:domain rdf:resource="#Selection"/>
</rdf:Property>

```

Figure 9 Partial Knowledge Base of the Sorting System

```

(and(or(http://www.w3.org/2000/01/rdf-schema#::label ?x "Adapter"
)(http://www.w3.org/2000/01/rdf-schema#::label ?x "Target" ))(http://
www.w3.org/2000/01/rdf-schema#::domain ?x ?y))

```

Figure 10 Query 1 of Property 1

```

(and (http://www.w3.org/2000/01/rdf-schema#::label ?x "sortStrategy" )
(http://www.w3.org/2000/01/rdf-schema#::domain ?x ?y))

```

Figure 11 Query 2 of Property 1

Property 2: every concrete product shall have an associated concrete factory that creates it in the Abstract Factory pattern.

This property can be expressed by the subtraction of the results of two queries. The first query, depicted in Figure 12, inquires all the classes which play the role of “ConcreteProduct” in the design with the query result stored in variable ?y. The second query shown in Figure 13 searches all the classes which have the “create” relationship and are created by other classes with the query result stored in variable ?y. These two queries can be inputted in the First goal and Second goal, respectively, in the web interface shown in Figure 7. The result of subtracting variable ?y of query 2 from variable ?y of query 1 is an empty set. From the query results, we can conclude that every concrete product has an associated concrete factory in the sort system design.

```

(and (http://www.w3.org/2000/01/rdf-schema#::label ?x "ConcreteProduct" )
(http://www.w3.org/2000/01/rdf-schema#::domain ?x ?y))

```

Figure 12 Query of Classes with Role “ConcreteProduct”

```

(and (http://www.w3.org/2000/01/rdf-schema#::label ?x "create_relation" )
(http://www.w3.org/2000/01/rdf-schema#::range ?x ?y))

```

Figure 13 Query of Classes with Relation “create_relation”

Property 3: In the Bridge pattern, the class that plays the role of Abstract shall be able to invoke the operations defined in the classes that play the roles of Implementor and Concrete Implementor.

After adding the selection sort algorithm in the design in Figure 8, we want to know whether all sort algorithms (Quick, Selection and Insertion) can sort character and number based on the comparison implementation in classes Character and Number, respectively. This property can be checked by a combination of two queries. The first query shown in Figure 14 searches all the classes which can compare numbers (has label “CompareNumber”) and their immediate invokers in the sort system design. Since classes Number, Library and SelectionLibrary can compare numbers, their caller, Implementor, Insertion, and Selection are returned as the result of the first query. The second query shown in Figure 15 gets the caller of the class Implementor. The caller is the Algorithm class. Similarly, the query in Figure 16 gets the caller of the Algorithm class and returns the caller Quick as the result of the second query. Combining the results of the two queries, we know that the classes Quick, Selection and Insertion can compare numbers. Comparing character can be checked in the same way.

```

(http://www.w3.org/2000/01/rdf-schema#::label ?x "CompareNumber" ) (http://
www.w3.org/2000/01/rdf-schema#::domain ?x ?class)(http://www.w3.org/2000/01/
rdf-schema#::label ?y "invoke" )(http://www.w3.org/2000/01/rdf-schema#::range
?y ?class)(http://www.w3.org/2000/01/rdf-schema#::domain ?y ?invoker)

```

Figure 14 Classes that can compare number and their caller

```

(http://www.w3.org/2000/01/rdf-schema#::label ?y "invoke" )(http://www.w3.org/
2000/01/rdf-schema#::range ?y [file:/F:/syang/JTP/sorting.rdf#::Implementor])(http://
www.w3.org/2000/01/rdf-schema#::domain ?y ?invoker)

```

Figure 15 The caller of the class Implementor

```

(http://www.w3.org/2000/01/rdf-schema#::label ?y "invoke" )(http://www.w3.org/
2000/01/rdf-schema#::range ?y [file:/F:/syang/JTP/sorting.rdf#::Algorithm])(http://
www.w3.org/2000/01/rdf-schema#::domain ?y ?invoker)

```

Figure 16 The caller of the class Algorithm

Now let us consider another evolution scenario of the sort system design as shown at the right-hand side of Figure 8. Besides Character and Number, suppose a new requirement of comparing the elements with the String type is added to the sort system. Based on the Bridge pattern, a new class String that plays the role of ConcreteImplementor, a child class of the Implementor class, is added into the system by a design team using our pattern evolver service shown in Figure 3. Since the Bridge pattern separates the abstraction from its implementation so that they can vary independently,

which means the addition of the String class shall be enough to allow all algorithms to be able to sort elements with the String type.

After adding the String class, we need to check system consistency using our checker service. All three properties are checked. Property 1 still holds because the evolution takes no action on the Adapter pattern. For Property 2, we perform the same check through our checker service shown in Figure 7; the query result returns the String class which indicates that the String class does not have a creation relationship associated. Property 2 does not hold any more after the evolution. The reason is that the String class also plays the role of ConcreteProduct in the Abstract Factory pattern. Thus, another class CreateString and a creation relationship from class CreateString to class String should also be added simultaneously into the sort system as shown in Figure 8. For the new sort system with the classes String and CreateString added, property 2 holds. In addition, we need to check property 3 and run the queries shown in Figure 14, Figure 15, and Figure 16 according to the evolution of the system, i.e., we need to check whether classes Quick, Insertion and Selection can sort the elements with the String type in the new system design. When we run the query in Figure 14 and replace “CompareNumber” by “CompareString” in the query, the query result only has the class Implementor, which means that the classes Insertion and Selection cannot compare the elements with the String type and property 3 does not hold for the new sort system.

4. Related Work

Tsai et al. [10] proposed a testing framework for verifying the completeness and consistency of web service specification by an algorithm. Both positive and negative test cases are generated. Modeling SOA and web service based on UML has been presented in [5][6]. By extending UML, the SOA modeling language captures requirements, creates extension solution, and optimizes these solutions [6]. A platform independent model for web service application is also developed [5]. Kim et al. [7] integrated service-oriented design and the concept of aspects in the design phase of software lifecycle to utilize services and aspects as fundamental and abstract elements. Lu [8] applied service-oriented architecture 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 a service-oriented architecture for the evolutions and analyses of design patterns at different locations.

5. Conclusions

As software industry becomes more and more globalized, the collaboration of the project teams from different sites becomes an important issue. Different tasks

of software development may be performed by the teams from different locations. The collaboration and management of the software development is a challenging issue. In this paper, we define a service-oriented architecture and provide the prototype tools for the evolutions and analyses of pattern-based designs. Our approach allows design teams to work together at different locations. We also present a case study to show our evolution processes and the checking of properties.

References

- [1] G. Booch, J. Rumbaugh, I. Jacobson. The Unified Modeling Language User Guide, Addison-Wesley, 1999.
- [2] 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, Germany, March 2006.
- [3] R. Fikes, J. Jenkins, and G. Frank, JTP: A System Architecture and Component Library for Hybrid Reasoning. *Proceedings of the Seventh World Multiconference on Systemics, Cybernetics, and Informatics*. Orlando, Florida, USA. July 2003.
- [4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- [5] C. He, F. He, K. He, W. Tu, Constructing Platform Independent Models of Web Application. *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, pp77-81, China, 2005.
- [6] Y. Jiang; C. Xing; W. He; J. Yang, On procedure strategy of constructing SOA's modeling language. *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, pp71-76, China, 2005.
- [7] T. Kim and C. Chang, Service-Oriented Design with Aspects (SODA). *Proceedings of the 2005 IEEE International Conference on Services Computing (SCC'05)*, pp. 319-322, Florida, USA, July 2005.
- [8] X. Lu, An Investigation on Service Oriented Architecture for Constructing Distributed Web GIS Application, *Proceedings of the 2005 IEEE International Conference on Services Computing (SCC'05)*, pp 191-197, July 2005.
- [9] W. T. Tsai, Service-Oriented System Engineering: A New Paradigm. *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, pp3-6, Beijing, China, 2005.
- [10] W. T. Tsai, X. Wei, Y. Chen, R. Paul, A Robust Testing Framework for Verifying Web Services by Completeness and Consistency Analysis. *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, pp151-158, Beijing, China, 2005.
- [11] ArgoUML, <http://argouml.tigris.org/>
- [12] W3C, Extensible Markup Language (XML), <http://www.w3.org/>
- [13] W3C, XSL Transformations (XSLT), <http://www.w3.org/>
- [14] Rational Rose website. <http://www.rational.com/>
- [15] JTP, <http://www.ksl.stanford.edu/software/JTP/>
- [16] RDF, <http://www.w3.org/RDF>
- [17] XMI2RDF, <http://freshmeat.net/projects/ju2d/>