

---

## Research

# Integrating diverse paradigms in evolution and maintenance by an XML-based unified model



Chih-Wei Lu<sup>1,2</sup>, William C. Chu<sup>3,\*</sup>, Chih-Hung Chang<sup>1</sup>,  
Don-Lin Yang<sup>1</sup> and Wen-Da Lian<sup>3</sup>

<sup>1</sup>Department of Information Engineering, Feng Chia University, Taiwan

<sup>2</sup>Department of Information Management, Hsiuping Institute of Technology, Taiwan

<sup>3</sup>Department of Computer Science and Information Engineering, Tunghai University, Taiwan

---

## SUMMARY

Many disparate software analysis/design methods and tools (called *paradigms* in this paper) individually offer faster and more efficient software design/evolution, yet they are generally partially incompatible. Therefore, when they are used together, they suffer from a lack of communication and integration. Unanticipated and inadvertent inconsistencies arising from the implicit incompatibilities result in significantly increased costs and errors during software maintenance. To improve both software evolution and maintenance, we propose an XML-based model that effectively unifies and integrates current software paradigms while helping reduce ripple effects and related software maintenance difficulties. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: life cycle; software models; software meta-models; model unification; model integration; ripple effects; XML

## 1. INTRODUCTION

Modern software systems need to be both flexible and efficient. The complexity of such systems can be readily seen in Figure 1 [1]. Each part affects the others, especially when changes occur either through the natural evolution of a program or through external changes such as user modifications or maintenance.

According to recent research and statistics [2–4], software maintenance consumes about 60–80% of the cost of a software system throughout its useful life. Software maintenance involves the procedures of system/program understanding, program modification, and ripple effect troubleshooting.

---

\*Correspondence to: William C. Chu, Department of Computer Science and Information Engineering, Tunghai University, No. 181, Sec. 3, Taichung Kan Rd, Taichung, 407, Taiwan, Republic of China.

†E-mail: chu@csie.thu.edu.tw

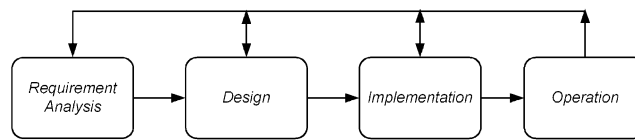


Figure 1. The typical process of the software maintenance life cycle.

Usually, understanding programs is hard work; the work may refer to documents scattered over the phases of analysis, design, and implementation. However, tracing the ripple effects of all document/artifact changes or program modifications is an even more complicated task since it needs to ensure the consistency of documents in different phases of the software life cycle, often based on the use of different methods and techniques. Without tool support and an appropriate approach, manually maintaining the accuracy and consistency of various documents in different phases of the software life cycle is very difficult, costly, and error-prone.

To reduce the high cost of maintenance and to facilitate software analysis, design, and evolution, many designers rely on a variety of widely recognized tools and methods. Based on industry standards [5–8], these tools and methods function as *paradigms* in design and implementation because they are reliable, fast, and efficient, using proven techniques and standard notations. For example, both *unified modeling language* (UML) [9] or *extensible markup language* (XML) [10,11] reduce the overhead of software inner-communication during the software life cycle and increase maintainability and reusability. Another well received method is provided through *design patterns* [12], which are reusable solutions to recurring problems in software analysis and design [4,13–20]. *Frameworks* likewise provide users with an efficient means to combine component reuse with design reuse [21]. *Component-based software engineering* (CBSE) also offers reusability as well as fast design/implementation, easy configuration, stable operation, and convenient evolution. Unfortunately, most systems are developed and maintained in an *ad hoc* manner with very limited enforcement of these commonly accepted paradigms. Such practices make software evolution and maintenance difficult and costly.

In the following section, we will show how these current methods and tools fall short of providing users with a truly reliable means of evolving and maintaining low-cost, efficient software systems. Next, we will introduce an XML-based model that effectively overcomes these problems. Then, we validate this approach with an implementation of this tool, which we tested by applying it to the persistent problems of ripple effects. In the fifth section, we will discuss this approach and the tool further with related experiments, comparisons, and some insightful studies. In our conclusion, we look toward the future prospects of implementing our approach.

## 2. CURRENT SOFTWARE PARADIGMS AND THEIR SHORTCOMINGS

The *paradigms* mentioned above usually cover only part of the software process. Figure 2 shows the relationship between these paradigms and a summary of the software maintenance life cycle [22].

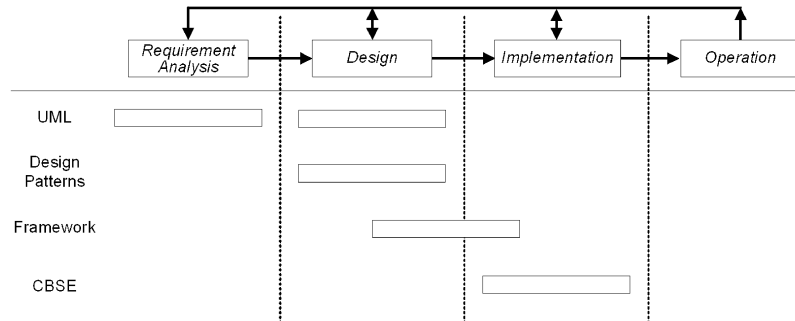


Figure 2. The relationship of some software paradigms and the software maintenance life cycle.

For instance, UML provides standard notations for modeling software analysis and design, yet it lacks support in the implementation phase. Design patterns, as the name implies, contribute primarily to the design phase, whereas CBSE focuses mainly on the implementation phase. Furthermore, frameworks benefit applications only in specific domains. Unfortunately, these paradigms are not ‘talking’ to each other, and therefore designers need to spend a lot of effort to map and integrate them in the various phases of the software life cycle. Without unifying and integrating these paradigms, the consistency of the models cannot be maintained, and the extent of automation must remain very narrow.

Modeling transfer and verification seek to overcome these problems. For example, modeling understanding helps an engineer compare artifacts by summarizing where one artifact (such as a design) is consistent or inconsistent with another artifact (such as source) [23]. Modeling verification, however, allows designers to analyze resulting diagrams in relation to their specifications using automated techniques. These techniques either validate behavior through simulation or check for numerous properties of the diagrams, including inter- and intra-model consistency [24].

In summary, none of these paradigms is general enough to cover all phases of the software maintenance life cycle. Thus, developers need to adopt more than one method or set of tools to accomplish their work; however, because most of the paradigms offer no connections to or compatibility with the others, gaps exist between these paradigms’ applications. Figure 3 shows where problems arise due to inconsistency between paradigms at different stages in the software life cycle. The notation  $\boxtimes$  expresses that there is a need for modeling transfer between successive phases/models in a specific paradigm; the notation  $\otimes$  points out the absence of consistency from one paradigm to another.

One of the most challenging problems of software maintenance is *impact analysis* (or *ripple effect analysis*) [25–27], which usually starts with a proposed modification to a piece of software code or design. The problem then causes a sequence of updates to the impacted models/documents scattered over various phases of the life cycle. Identifying the related components of the system involved with the impact of the proposed modification is difficult and costly since the models/documents/codes are often developed heterogeneously and inconsistently.

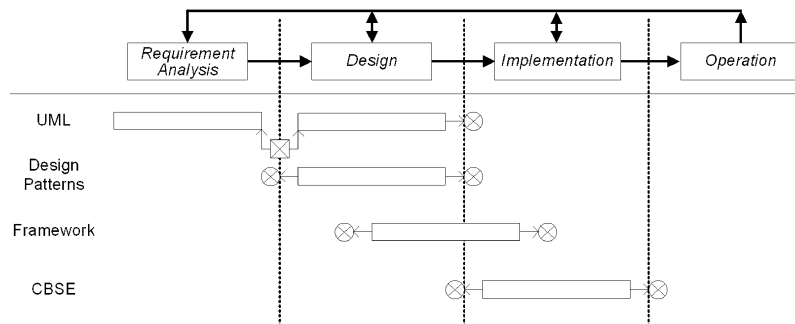


Figure 3. Problems of inconsistency between paradigms and the software maintenance life cycle.

Attempts to deal with such problems in a *post hoc* fashion seem to mirror the *ad hoc* manner in which systems are often developed. For example, current semantic analysis approaches are based mainly on programming languages. They offer limited assistance and imprecise information for maintenance purposes. As a result, revisions for reliability and consistency can end up being risky and unpredictable. We are left with spending great amounts of time and effort to solve what in effect are ‘built-in’ problems.

We believe that it is possible to greatly improve the maintainability of software systems by integrating various models with different paradigms throughout the various phases of the software life cycle. To achieve that end, we propose a unified system model—the basis for which is found in XML. In the following section, we show how our XML meta-model unifies and integrates the existing software paradigms and thereby improves software evolution and maintenance.

### 3. AN XML-BASED UNIFIED META-MODEL (XUMM)

Getting current and future paradigms to work together and provide synergies across all the phases of the software life cycle may seem a daunting challenge. We believe, however, that there is already a kind of ‘proto-type’ for our unified model in XML. For clarity, we feel we should briefly recount the features of XML.

#### 3.1. Extensible markup language (XML)

First, XML [28] is a standard language supported by W3C (World Wide Web Consortium). Second, it offers application neutrality (vender independence), user extensibility, ability to represent arbitrary and complex information, validation for data structure scheme, and human readability. Third, it has a number of widely known and well-tested components:

- DTD (document type definition), which specifies a set of tags and their constraints. The format and interpretation of XML documents are defined by DTDs.



- XSL (extensible style sheet language), which describes style sheets for an XML document. That is, it covers the document presentation. XSL is based on DSSSL (document style semantics and specification language) and CSS (cascading style sheet).
- Xpointer and Xlink, which define anchors and links across XML documents.

Within XML is *XML schema*, a language that defines structure and constraints of the contents of XML documents. An XML schema consists of a set of type definitions and element declarations. These can be used to assess the validity of well-formed elements and attribute information items; they may also specify augmentations to those items and their descendants. The primary components of an XML schema are as follows:

- Simple type definitions, which cannot have element content nor carry attributes.
- Complex type definitions, which allow elements as their content and may carry attributes.
- Attribute declarations, which are associations between a name and a simple type definition, together with occurrence information and (optionally) a default value.
- Element declarations, which are associations of a name with a type definition, either simple or complex, a (optional) default value and a (possibly empty) set of identity-constraint definitions.

Suzuki *et al.* [29] proposed an XML-based UML exchange format (UXF) to transform UML diagrams into XML. However, they focused on the format exchange between UML and XML representations, whereas we seek to integrate and unify software paradigms under one meta-model.

### 3.2. XUMM model overview

In our approach, we use an *XML-based unified meta-model* (XUMM) to define the schema of an *XML-based unified model* (XUM). This allows us to consolidate and coordinate modeling information from the adopted paradigms—such as analysis and design models represented in UML, design patterns, framework, etc.—in each phase of the software life cycle.

To avoid confusion with the various uses of the term ‘model’, we refer in this paper to those models composed with a paradigm as ‘sub-models’ of our integrated, unified model. We call them sub-models because each one characterizes the system partially, in one aspect of a specific phase. Through the transformation of XUMM, a sub-model can be transformed into its corresponding XML representation, which we call a ‘view’ of the XUM.

As shown in Figure 4, based on XUMM, sub-models are unified, integrated, and represented as views of an XUM. Semantics in each sub-model should be described explicitly and transferred precisely in XUM.

The XUM facilitates the following tasks:

- capturing of modeling information of models and transforming into views of XUM;
- two-way mapping of modeling information among models and XUM views;
- integration and unification of modeling information of different views in XUM;
- support of systematic manipulation;
- assisting the consistency checking of views represented in XUM; and
- reflection of changes of view information in XUM to models in each phase.

The details of XUMM as well as XUM will be discussed in the following sections.

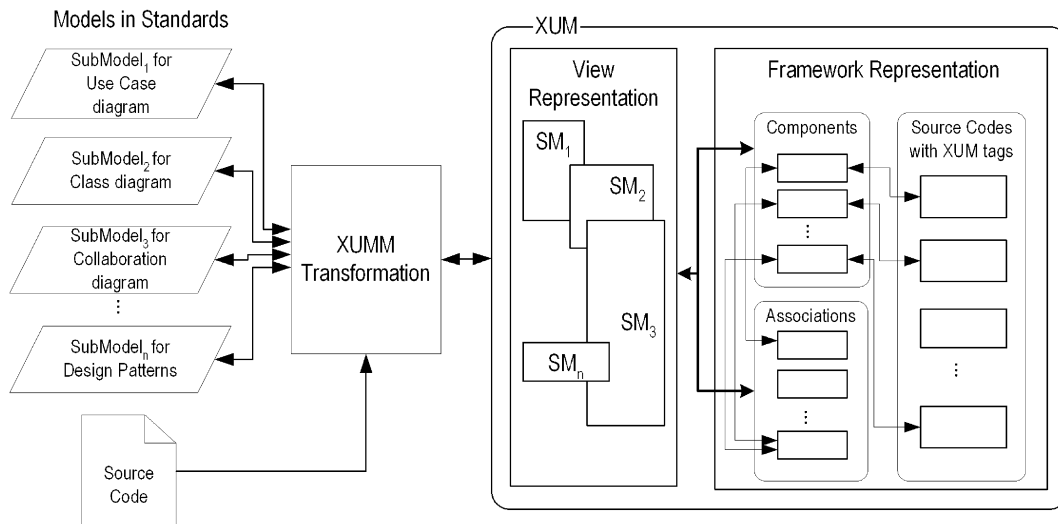


Figure 4. The unification and integration of models into XUM.

### 3.3. XUMM model details

Figure 5 shows the relationship of views in XUM. The major merits of XUM are (1) the modeling information used in models (views) of each phase of the software life cycle and (2) the interaction and relationship of models (views). Both are explicitly defined and represented in XUM.

The relationship of the XUMM with an XUM is similar to that of a DTD with an XML document. XUMM defines the schema (definitions) of an XUM. Three kinds of elements defined in XUMM describe the constitution of an XUM; they are *component*, *association*, and *unification relation*. Any object in an XUM is identified as a component. Components and associations describe the semantic information of model objects and their relationships, respectively. The third kind of element, unification relation, describes the relationship of different views.

According to the three kinds of elements, three primitive schemas are defined in XUMM—*ComponentType*, *AssociationType*, and *UnificationLinkType*. The *ComponentType* schema defines the necessary modeling semantic information and the types that are used to describe components in our unified model. The XML schema of *ComponentType* is defined as follows:

```
<xs:complexType name="ComponentType">
  <xs:sequence>
    <xs:element name="id" type="xs:string"/>
    <xs:element name="name" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

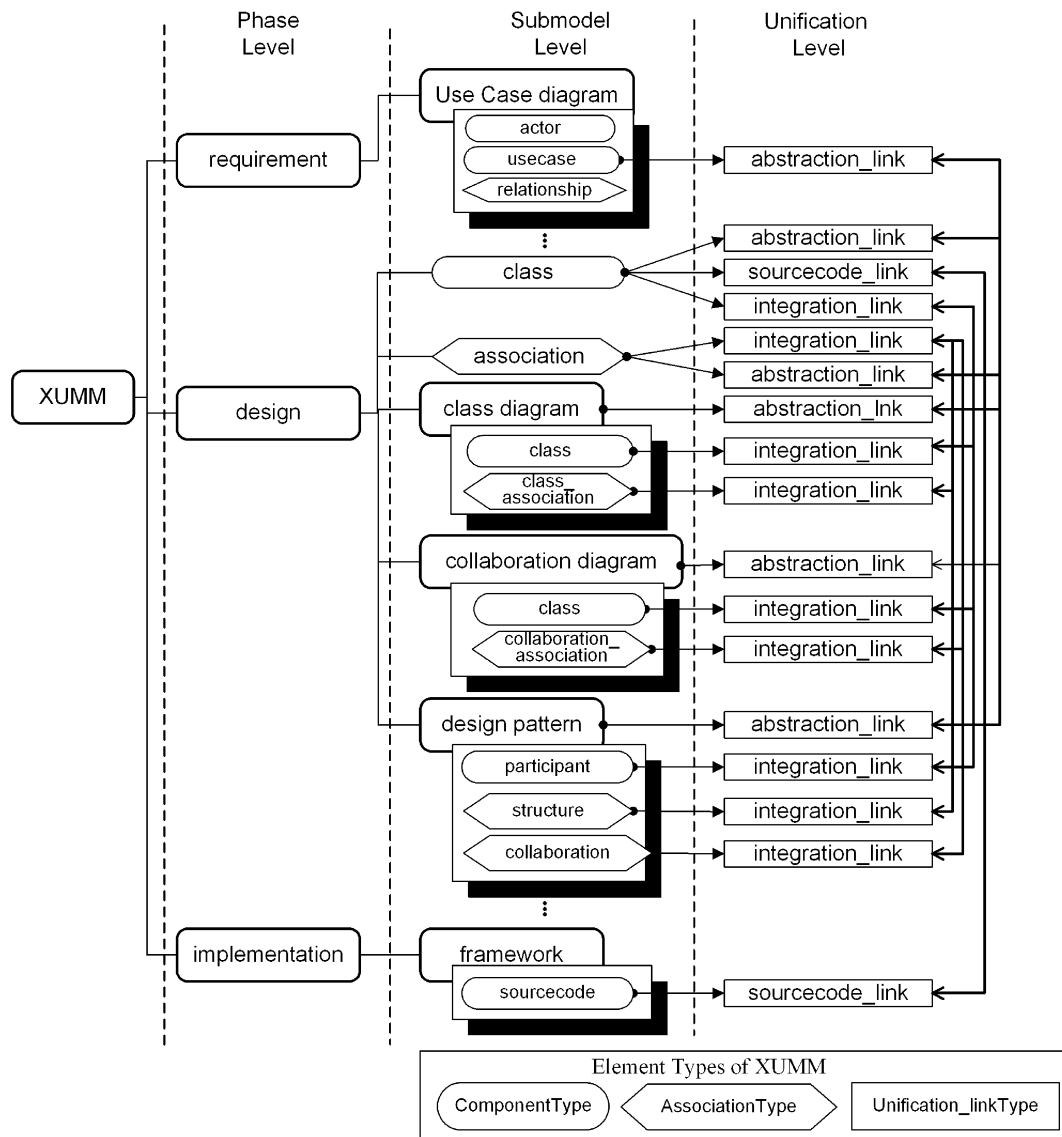


Figure 5. The relationship of views in XUM.



The *AssociationType* schema defines the necessary information and the types that are used to describe the relationships of components. The XML schema of *AssociationType* is defined as follows:

```
<xs:complexType name="AssociationType">
  <xs:sequence>
    <xs:element name="link_id" type="xs:string"/>
    <xs:element name="from" type="xs:string"/>
    <xs:element name="to" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

In order to show the relationship of the integration and unification of views in XUM, *UnificationLinkType* is defined. *UnificationLinkType* schema defines the hyperlink relations between elements in an XUM using a set of *xlinks*. The XML schema of *UnificationLinkType* is defined as follows:

```
<xsd:complexType name="Unification_linkType">
  <xsd:annotation>
    <xsd:documentation>Definition of primitive element:
      Unification_link</xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:attribute name="xlink:type" type="(locator)" use="fixed" value=""/>
    <xsd:attribute name="xlink:arcole" type="xsd:Cdata" use="required"/>
    <xsd:attribute name="xlink:href" type="xsd:string" use="required"/>
    <xsd:attribute name="xlink:title" type="xsd:Cdata"/>
    <xsd:attribute name="xlink:from" type="xsd:NMTOKEN"/>
    <xsd:attribute name="xlink:to" type="xsd:NMTOKEN"/>
  </xsd:complexType>
</xsd:complexType >
```

Based on the purposes of *UnificationLinkType*, three types of links are defined further—*integration\_link*, *abstraction\_link* and *sourcecode\_link*. The *integration\_link* is used to link a set of components and/or associations that have the same semantics but may be named or represented differently in different views. The XML schema of *IntegrationLink* is defined as follows:

```
<xs:element name="integration_link" type="Unification_linkType"/>
```

The *abstraction\_link* is used to link a component/association to a view. The view consists of a set of components and their associations; it also represents the details of a specific component at a lower level of abstraction. The XML schema of *AbstractionLink* is defined as follows:

```
<xs:element name="abstraction_link" type="Unification_linkType"/>
```

The *sourcecode\_link* is used to link a component to its corresponding source code. The XML schema of *SourcecodeLink* is defined as follows:

```
<xs:element name="sourcecode{\\_}link" type="Unification{\\_}linkType"/>
```



Table I. Mapping of model elements and XUM elements.

Models/paradigms	Model elements	XUM element representations
UML Use case diagram	Actor	<actor>
	Use case	<usecase>
	Association	<relationship type="association">
	Generalization	<relationship type="generalization">
	Extend	<relationship type="extend">
UML Class diagram Collaboration diagram Sequence diagram	Include	<relationship type="include">
	Class	<class>
	Attribute	<attribute>
	Operation	<operation>
	Interface	<interface>
	Parameter	<parameter>
	Association	<association type="association">
	Composition	<association type="composition">
	Generalization	<association type="generalization">
	Dependency	<association type="dependency">
Design patterns	Message	<message>
	Participants	<role>
	Structure	<structure>
	Collaborations	<collaboration>

Based on the integration, abstraction, and sourcecode links, the sub-models—adopting various paradigms that might share some semantics but were not explicitly represented—can be integrated and unified in XUM. Therefore, when a sub-model (view) is changed, the changes can be transferred to other related sub-models (views).

Each sub-model has its corresponding XUM representation, the view, and its schema defined in XUMM. Following the transformation of XUMM, transforming modeling information of a sub-model into a view of the XUM is not a difficult task. Due to the space limitation, we show only the structure of XUMM in Figure 6 and its mapping rules in Table I. The detailed schema of XUMM is shown in the Appendix.

### 3.4. XML-based unified model (XUM)

As mentioned above, our XUM is a representation of artifacts of software systems defined in XUMM. These artifacts are the modeling information collected from sub-models of paradigms used in each phase of the software life cycle.

First, each sub-model is transformed and represented as a view in XUM. The semantics of sub-models are explicitly captured and represented in views of XUM.

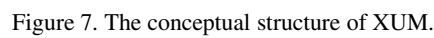


```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:complexType name="ComponentType">
    ...
    <xs:complexType name="AssociationType">
      ...
      <xs:complexType name="Unification_linkType">
        ...
        <xs:element name="xumm">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="requirement">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element ref="actor" minOccurs="0" maxOccurs="unbounded"/>
                    <xs:element ref="usecase" minOccurs="0" maxOccurs="unbounded"/>
                    <xs:element ref="usecase_diagram" minOccurs="0" maxOccurs="unbounded"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="design">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element ref="design_class" maxOccurs="unbounded"/>
                    <xs:element ref="class_diagram" minOccurs="0" maxOccurs="unbounded"/>
                    <xs:element ref="collaboration_diagram" minOccurs="0" maxOccurs="unbounded"/>
                    <xs:element ref="design_pattern" minOccurs="0" maxOccurs="unbounded"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="implementation">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element ref="framework"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="unification link">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element ref="integration link" minOccurs="0" maxOccurs="unbounded"/>
                    <xs:element ref="abstraction link" minOccurs="0" maxOccurs="unbounded"/>
                    <xs:element ref="sourcecode link" minOccurs="0" maxOccurs="unbounded"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

Figure 6. The structure of XUMM.



*J. Softw. Maint. Evol.: Res. Pract.* 2003; **15**:111–144



Last, the manipulation of views of XUM is accomplished through XML's published interfaces based on the document object model (DOM). That is, DOM is the internal representation of XUM. Therefore, the systematic manipulation of XUM can be accomplished through the manipulation on DOM of XUM.

The unification link plays a very important role in tracking the related elements that reside in different views. These related elements may have abstraction relations, which are connected by *abstraction\_links*. Similarly, *integration\_links* connect views that share the same elements. *Sourcecode\_links* do the same for components in views that are related to source codes. Based on these links, if any information in each view or any source code gets changed, the affected views can be reflected by following the links.

During software maintenance, modification to any sub-model should be detected and should reflect the effects on the related sub-models; the semantics in each sub-model can then be updated appropriately according to the modification. This assists the consistency checking of modeling information of views. In addition, impact analysis can be applied to the entire software system, including the impact on related source codes, the impact on related design information, and the impact on related requirement information.

#### 4. IMPLEMENTATION AND EXPERIMENTS

In the next section, we explain the implementation of the prototype XUM-based Model Integration Tool (XUMIT). In subsequent sections, we test our model against various ripple effects. These experiments facilitate and validate the features that are promised by the XUM approach.

##### 4.1. XUMIT implementation and its features

As shown in Figure 8, the XUMIT has adopted a model-view-controller (MVC) design pattern as the system architecture; each view will be automatically reflected when its corresponding model has been changed.

Java is the implementation language of XUMIT. Therefore, it can be run on different OS platforms such as Windows, Linux, or UNIX. Figure 9 shows the window interfaces of the tool set of XUMIT.

Briefly, there are five major, collaborative tools in XUMIT:

- (A) The *XUM structure browser*, to list all sub-models and elements of the subject system in XUM structure.
- (B) The *model viewers*, to display the contents of the corresponding sub-models with the formats of the software paradigms that users have selected from XUM browser or any other windows.
- (C) The *maintenance log viewer*, to log each effective action and then manage the status of the correlated ripple effects from those actions in the maintenance process.
- (D) The *ripple effect analyzer*, to analyze and reveal all the modules within the ripple effects that are caused by an action logged in the maintenance log recorder.
- (E) The *XUM query browser*, to filter and locate the XUM elements with the requests and specifications assigned by users.

To show more features of XUMIT, we have chosen a College Information Management System (called CIMS) as a subject system that will be maintained in XUMIT. CIMS is designed to manage

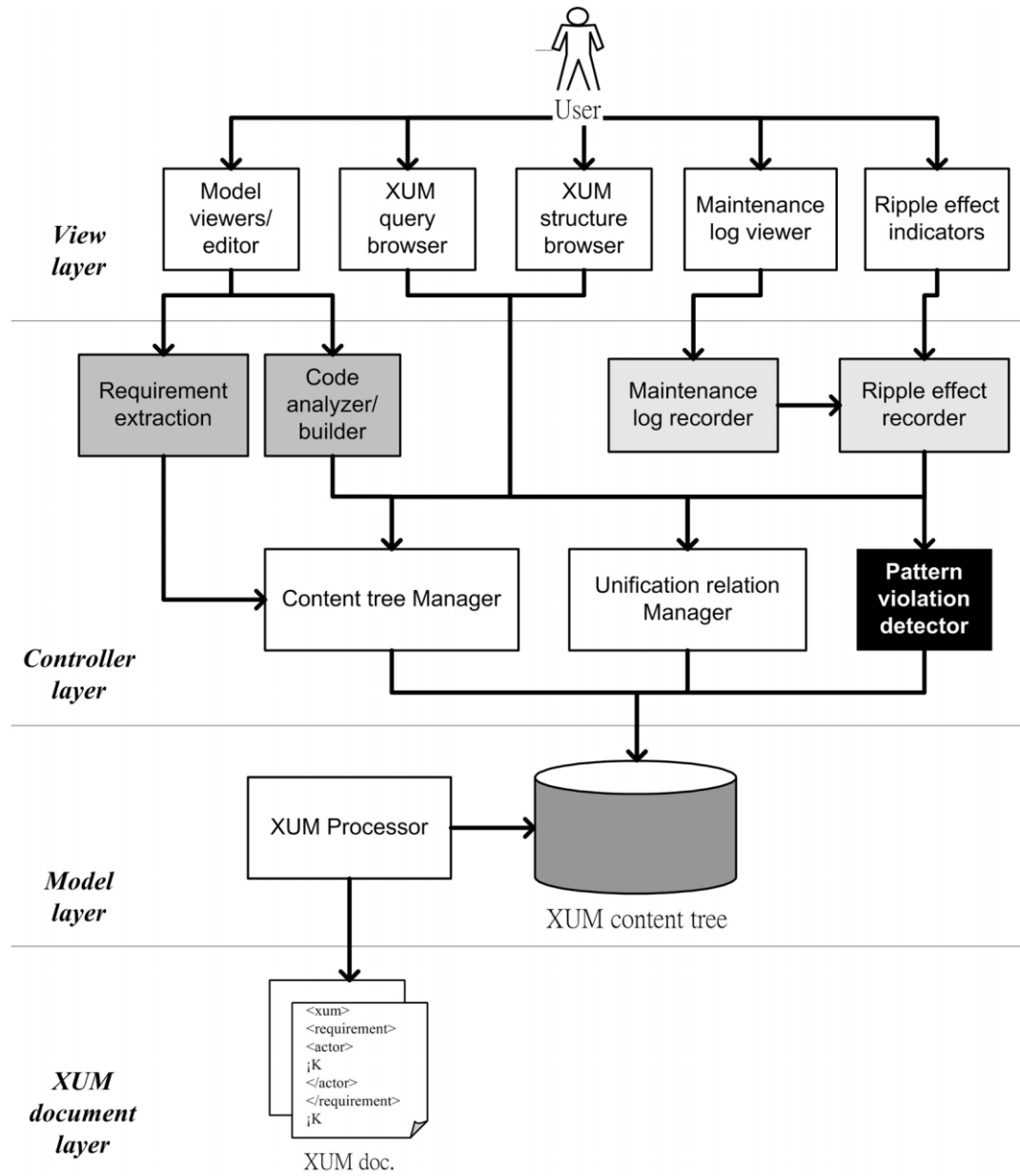


Figure 8. The system architecture of the tool XUMIT.

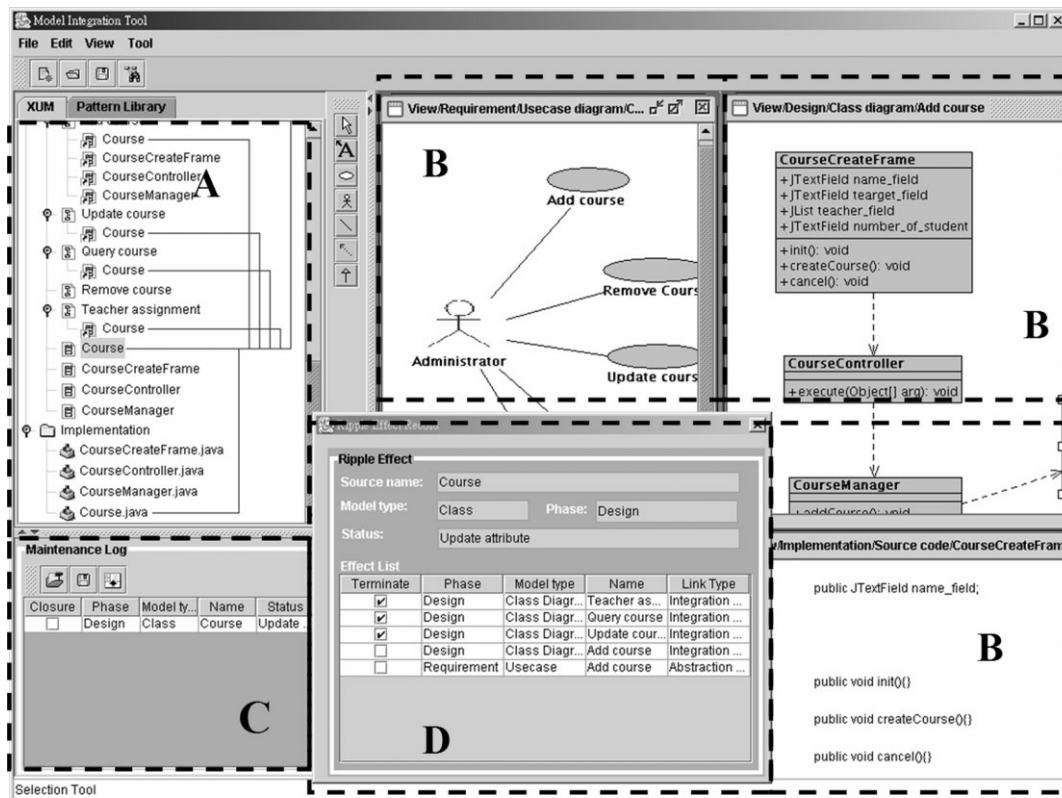


Figure 9. The layout of the interfaces of the tool XUMIT.

course-related information and activities, including Courses, Credits, Students, and Teachers. It also includes related subsystems such as college group communication. CIMS is a window-based system implemented in the Java language with 157 classes and a size of 14 103 lines of code. Table II shows selected statistics describing this system.

To illustrate how XUM/XUMIT unifies and integrates the sub-models of a system with various paradigms, we will show three demonstrations. Each demonstration shows how XUM/XUMIT tracks and solves different categories of ripple effects and how well it can maintain system consistency. We apply XUM to the following kinds of ripple effects:

- model-to-model (M2M) ripple effects;
- standard-to-standard (S2S) ripple effects; and
- phase-to-phase (P2P) ripple effects.



Table II. A detailed listing of the subject system.

Phases	Model types	Model count (total: 372 models)
Requirement	<i>Use cases</i>	
	Add teacher	
	Update teacher	
	Query teacher	
	Remove teacher	
	.....	28
	<i>Use-case diagrams</i>	
	Teacher management	
	Student management	
	Course management	
	Grade management	
	Group communication	
	Document management	
	Announcement	
Design	Discussion	
	.....	8
	<i>Classes</i>	
	Teacher	
	Student	
	Course	
	TeacherCreateFrame	
	TeacherEditorFrame	
	TeacherQueryFrame	
	CourseManager	
	StudentList	
	.....	157
	<i>Class diagrams</i>	
	Add teacher	
Implementation	Update teacher	
	Query teacher	
	.....	22
	<i>Source code modules (lines of code)</i>	
	Teacher.java(70 LOC)	
	Student.java(197 LOC)	
	Course.java(181 LOC)	
	TeacherCreateFrame (267 LOC)	
	.....	157 (14 103 LOC)



#### 4.2. Applying XUM to model-to-model (M2M) ripple effects

The CIMS allows users to establish, update, and query course information. When a change request comes from the user of CIMS, our first experiment shows how XUMIT can assist this change process. In this case, a teacher's qualification information is required in order to add the course information. This involves adding a new attribute, 'teacher\_qualification', to the Course class. Adding a new attribute to an existing class is a frequent activity of software maintenance.

As shown in Figure 10, in the first step XUM query browser is used to locate and bring up the window of the related class diagram, 'Add course', which contains the class 'Course'. In the second step the new attribute 'teacher\_qualification' is then added to the class 'Course' in the desired class diagram 'Add course'.

With any update or change to the component of a software system maintained by XUMIT, the update information will be kept in the maintenance log. Any change to a component of the system may cause ripple effects to the other components. *Maintenance log viewer* is designed to reveal the history information of any update. As demonstrated in Figure 11, from the log information, one click on a selected component, the components 'ripple-affected' by the selected component will be revealed in the ripple effect analyzer window. Any sub-model or component shown in analyzer window can be magnified to reveal more detailed information. Since XUM has kept the relationship of the related components, ripple effect analyzer can track all the sub-models that are related to or affected by the update of the class 'Course'.

With the help of XUMIT, users can have a clear picture of the impact of any change, and ripple effects can be automatically revealed. For example, the class 'Teacher\_assignment' is one that needs further attention due to the change to class Course. Therefore, a class structure consistency update can be performed very easily in comparison to updates without the assistance of XUMIT.

In order to have change consistency, every maintenance change usually causes a set of necessary adjustments in the other sub-models; however, each additional adjustment results in another new cycle of ripple effects. Figure 12 shows an example of possible further modification. XUMIT deals effectively with this problem in two ways: (1) it logs every update action; (2) it monitors and ensures that each set of the ripple effects is closed up, leading to a complete convergence. Figure 13 shows the process of the convergence of the ripple effects in XUMIT.

As demonstrated in the case above, when a class/sub-model is changed, the integration and unification of XUM characteristics allows for the tracking of all related sub-models; XUM also delivers the necessary extra updates. In short, our XUM approach has the capability to sustain the system consistency of model-to-model (M2M) ripple effects. In the next section, we will show how XUM can be used to uphold the consistency of standard-to-standard (S2S) ripple effects.

#### 4.3. Applying XUM to standard-to-standard (S2S) ripple effects

In the second demonstration, we determine if XUM can help maintain the consistency of components across different paradigms, which involve different types of software methods, tools, and industry standards. For simplicity, we refer to the wide-ranging problems that can result from inconsistencies among paradigms as standard-to-standard ripple effects.

In the subject system, another well-adopted paradigm, design patterns, has been used for design and implementation. The design pattern, 'Mediator', has been applied in Course Register. We assume that

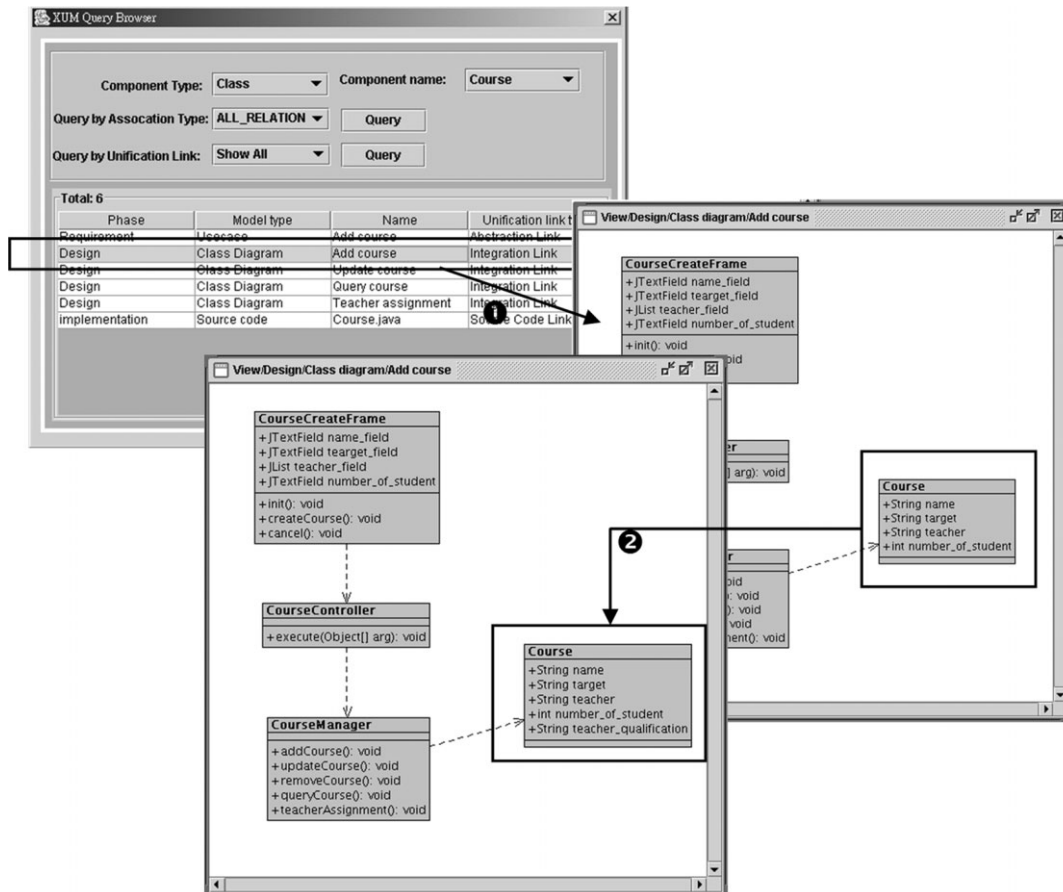


Figure 10. To browse and locate a related class diagram, then add a new attribute to an existing class.

a new function needs to be added to Course Register in order to notify students about the courses they have chosen. Following the procedures above, we locate the target class diagram 'Course Register'. Next, we place the desired new class 'Notification' in the design. Finally, we use a prompt to assign it to the role of 'Colleague' in the pattern 'Mediator', as shown in Figure 14.

Using OOA/D (object-oriented analysis and design), the maintainers may decide to associate the new feature class 'Notification' with the data class 'Course' using a class relationship of dependency, as displayed in Figure 15.

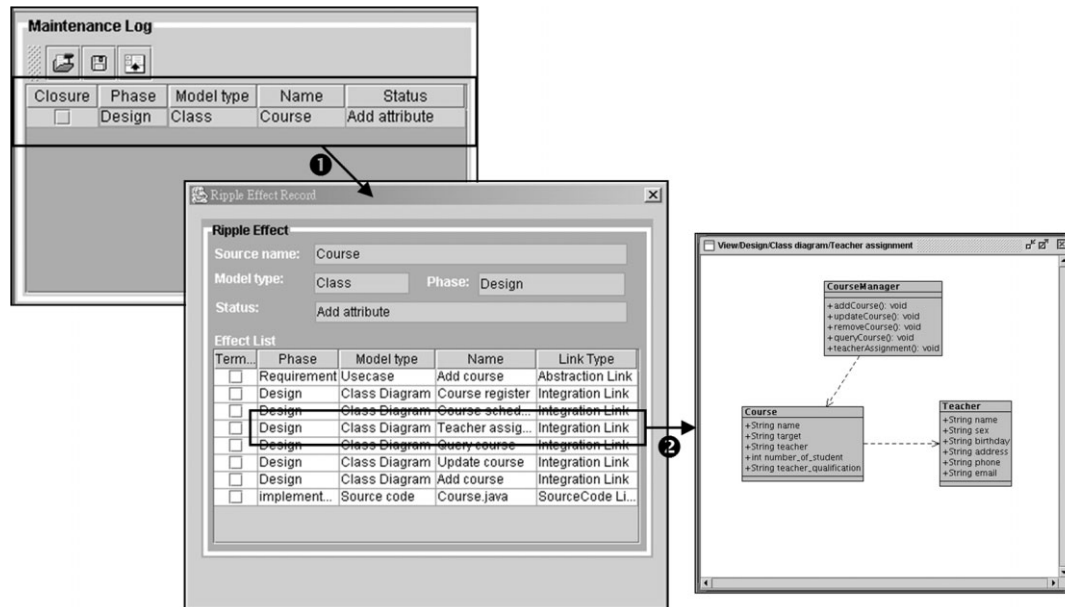


Figure 11. The XUMIT maintenance log and the corresponding ripple effect management.

The updates shown above conform to the general concept of OOA/D and to the design of the class diagram itself. However, within the constraints of the design pattern (which are defined in the specifications of the design pattern definition) there are at least two violations of the rules that a Mediator pattern should follow: (1) there should be one direct dependency relationship from the mediator to each of the colleagues; and (2) there should be no self-dependencies among any of the colleagues. Without the support of XUM, it is difficult and time consuming to detect this kind of inconsistency, especially for sub-models that belong to various paradigms or to different levels of abstraction. Figure 16 shows the details of the XUM specifications for the constraints of a Mediator pattern, which define the constraints between the two roles 'Mediator' and 'Colleague'.

XUM makes up for that inadequacy. Figure 17 exhibits the effect of XUMIT. In the example, while checking the improper update action from the maintenance logs, the constraint violations will be detected and revealed in the *model viewer* of the design pattern definition. Figure 18 shows the modification to the improper design and reveals how XUMIT maintains the consistency between different types of paradigms.

In short, our XUM approach assists and guides users in solving the problems of both S2S and M2M ripple effects. In the next section, we will show how our approach also overcomes phase-to-phase effects.

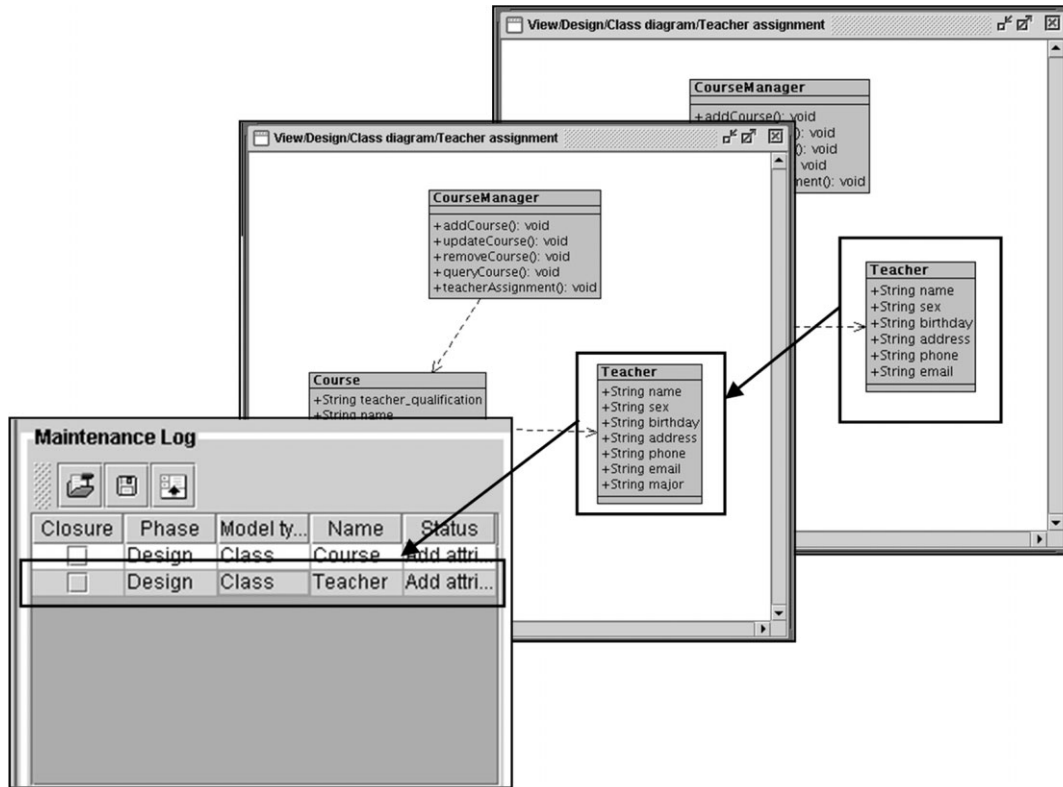


Figure 12. A class structure update for consistency corresponding to the ripple effect analysis in XUMIT.

#### 4.4. Applying XUM to phase-to-phase (P2P) ripple effects

Since there are no formal barriers to the sub-models in the phases for software analysis and design, we can extend the capability of XUMIT to achieve consistency across different sub-models of different phases.

The third case is virtually the same as the one used in Section 4.1. The difference is that in this situation, we start the update from the point of the corresponding source code module 'Course.java', as shown in Figure 19.

In XUMIT, updates to the class attributes on a source code module are reflected in the corresponding class 'Course' automatically, as shown in Figure 20. Following procedures similar to those stated in Section 4.1, the sub-models related to that updated class are collected in the ripple effect recorder, including the ones from the design phase and the requirement analysis phase. Figure 20 shows in details how XUMIT can assist users to perform the corresponding maintenance. By following the maintenance

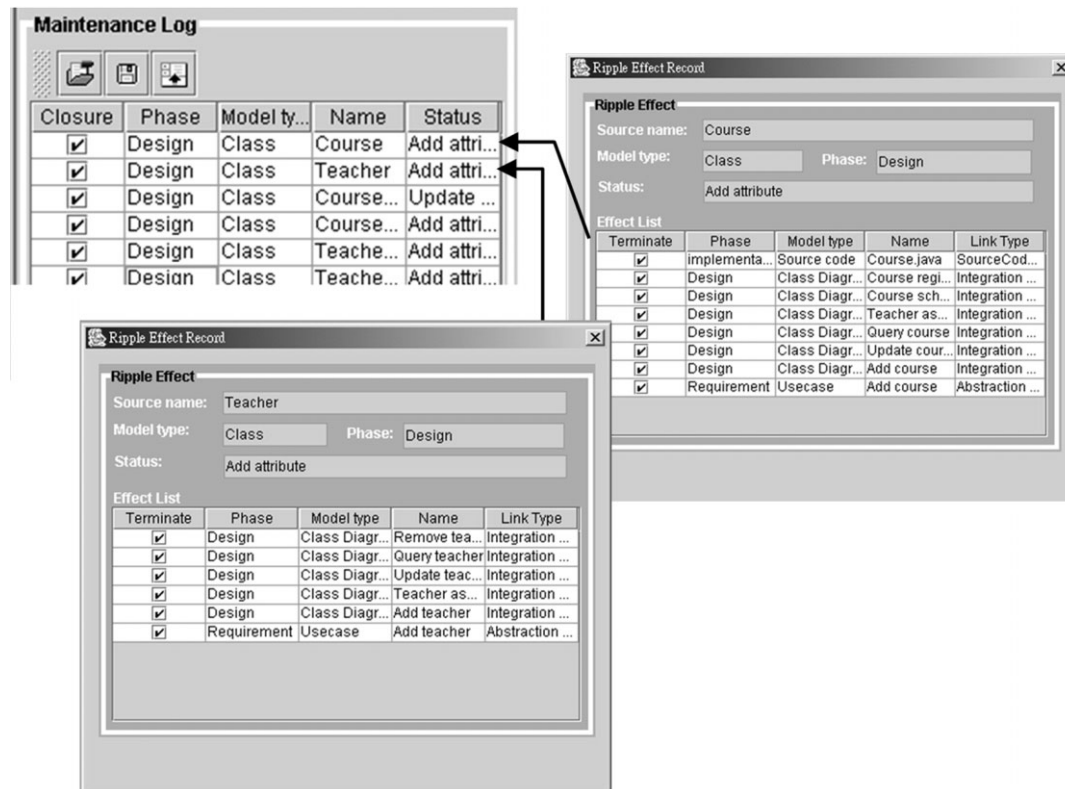


Figure 13. The convergence of ripple effects in XUMIT.

logs of updating the source code 'Course.java', we can trace the ripple effect for the update of the class 'Course', as well as the related sub-models in the requirement analysis phase. Then, in order to be consistent with the update, a new precondition, 'teacher\_qualification', is added to the specifications of the desired use case 'Add course', with the guidance of XUMIT.

As we can see in this third case, XUMIT maintains the consistency from the source code update to the corresponding class diagram changes, as well as to the adjustments of the use case. The encouraging results of the experiments above have demonstrated the merits of our XUM approach in helping to maintain system consistency between sub-models and paradigms and through all software phases.

According to the demonstrations shown in previous sections, the XUM approach is able to aid the processes of software maintenance practically and effectively. In the following section, we perform further experiments and comparisons to disclose the effectiveness of XUM.

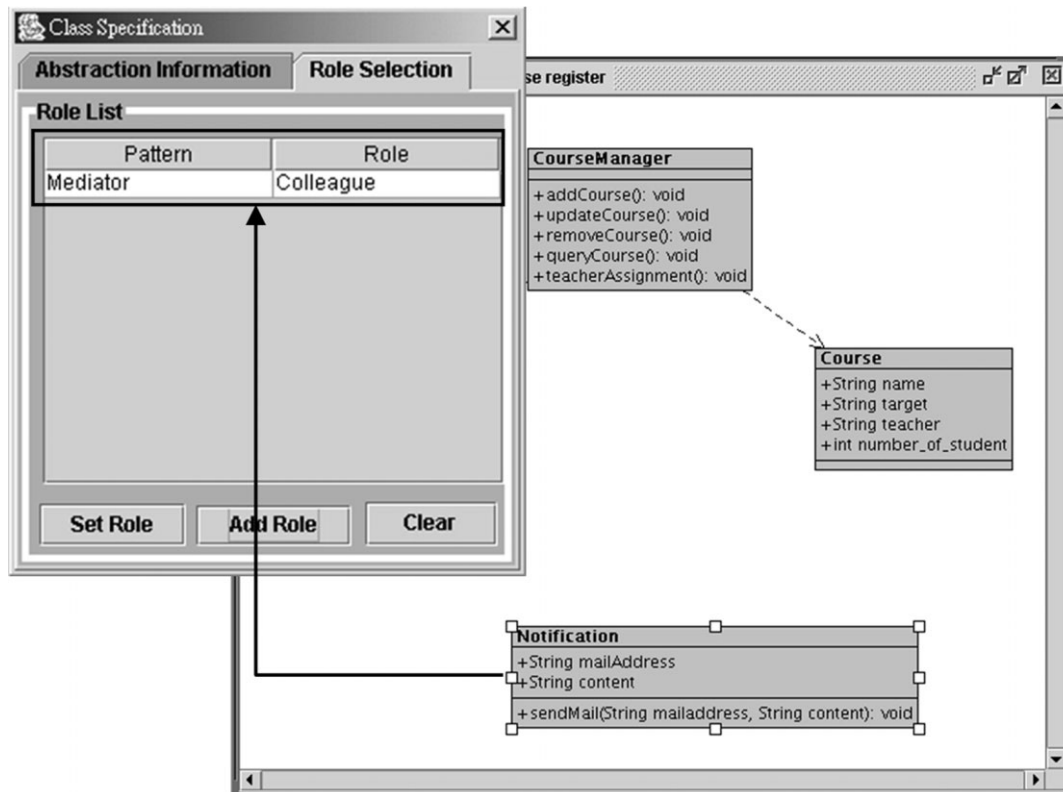


Figure 14. Adding a new class in the class diagram with an assignment as a role of a Mediator pattern.

## 5. COMPARISONS AND FURTHER DISCUSSIONS

### 5.1. Comparisons between using XUMIT and the traditional way

In order to assess the effectiveness of XUM, we assigned the three cases above to two different maintainers. One maintainer had the assistance of XUMIT, and the other followed the traditional approach only.

For the comparison, we adopted three factors: completeness, precision, and time-cost. We used the factor *completeness* to show how many maintenance case models could be located accurately. We used the factor *ripple-coverage* to display how many models maintainers have to scan/parse through ripple effects in order to gather all the effective updates they have done. We used the factor *time-cost* to present the sum of the time needed to locate the candidate models and perform the corresponding repairs. Table III shows how each maintainer fared.

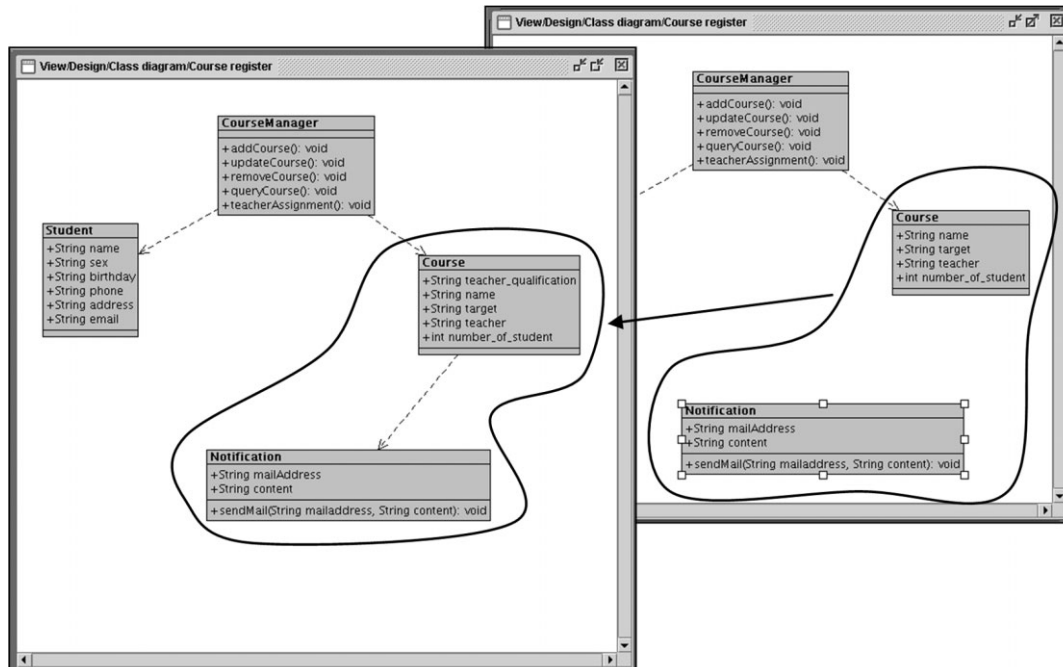


Figure 15. A newly added class relationship, a dependency from 'Course' to 'Notification'.

The experiments shown in Table III demonstrate clearly the advantages of using XUMIT over the traditional approach to maintenance. For *completeness*, the experiment shows that a maintainer using the traditional way can miss as many as half of the models that should easily be found and fixed. However, with XUMIT, all the implicit dependencies and sharing relationships are revealed automatically. Thus, users can track all the related models by following the directions. As for *ripple-coverage*, maintainers using the conventional method need to parse all the models of a system for each round of the impact analyses. In contrast, XUM dramatically improves the efficiency of maintenance by drastically reducing the number of models that need to be handled through the ripple effects; it presents to the users the related models only. Lastly, considering the factor of *time-cost*, we can confidently claim that the assistance of XUM/XUMIT saves most of the 'man hours' of maintenance, for it automates the complicated procedures of locating the scattered models in ripple effects, and it does so with different paradigms and across the different phases of the software life cycle.

## 5.2. Further discussions for XUM in the experiments

In addition to the discussion above, there are three points that should be discussed further in the case study. First, the way to capture modeling information from sub-models and then transform them



```
<design_pattern>
  <pattern>
    <id>23756:ecaebe8ea0:-7ffa</id>
    <name>Mediator</name>
    ... ..
    <constraint>
      <role_from>Colleague</role_from>
      <constraint_rule>no depend on</constraint_rule>
      <role_to>Colleague</role_to>
    </constraint>
    <constraint>
      <role_from>Mediator</role_from>
      <constraint_rule>associate</constraint_rule>
      <role_to>Colleague</role_to>
    </constraint>
    <constraint>
      <role_from>Colleague</role_from>
      <constraint_rule>associate by</constraint_rule>
      <role_to>Mediator</role_to>
    </constraint>
    ... ..
  </pattern>
</design_pattern>
```

Figure 16. The XUM specifications for the constraints of a Mediator pattern.

into view representations in an XUM is quite systematic and straightforward as long as the mapping rules between two representations are well defined in XUMM. In our approach, each sub-model that adopts a software paradigm should have its corresponding view representation. The views carrying and sharing information from the global information repository—the XUM—can explicitly and completely define the semantics of components and their relationships, which may be implicitly or incompletely represented in the paradigm sub-models.

Second, in addition to the transformation from a sub-model to a view, it is necessary to keep the two-way mapping in an XUM between the sub-models and their views in order to project a paradigm model as needed. In an XUM, the naming of elements in views is the same as that in the corresponding sub-models; therefore, two-way mapping can be achieved.

Third, as shown in the XUM representation in the previous section, the unification links connect the components and associations that share some semantic information. When modeling information is not complete, some of these links may be undefined. However, these undefined links are very valuable to software engineers, for they indicate that the system is incomplete and requires enhancements.

## 6. CONCLUSION

As we have seen, current software paradigms are necessary but not sufficient to meet all the needs of software evolution and maintenance. None of the methods and tools we investigated covers all

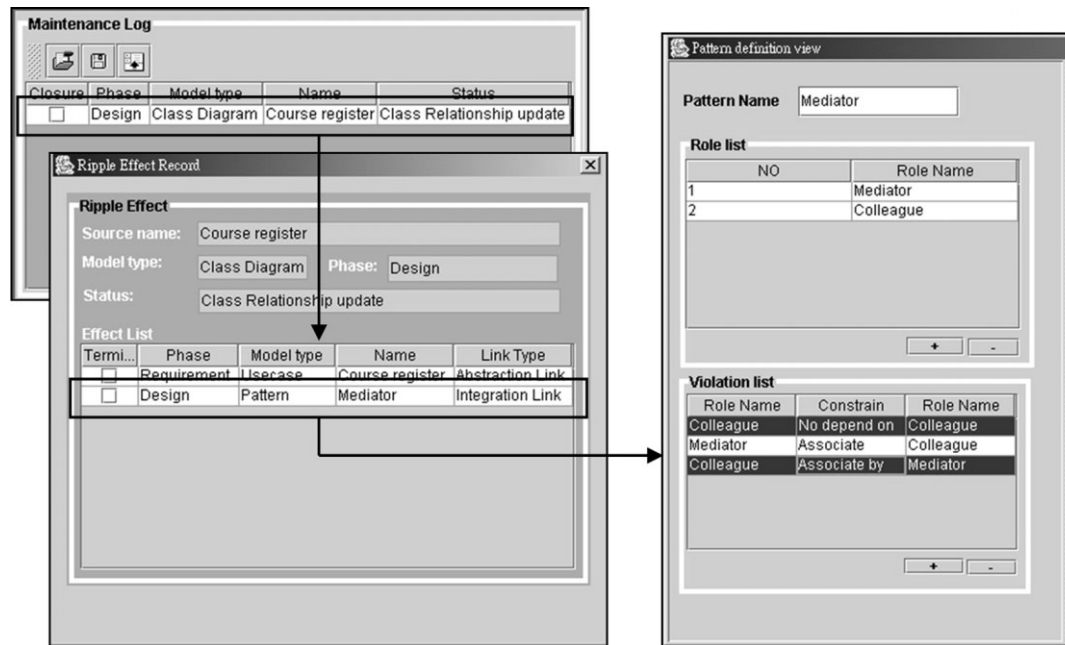


Figure 17. An example of inconsistency and violation between a class diagram and a design pattern.

the phases of the software maintenance life cycle, and few of them are compatible with each other. Using these 'isolated' paradigms engenders a lack of integration and consistency. This especially affects the more serious problems of maintenance in all aspects.

In this paper, we have proposed an XML-based unified model, called XUM, which can integrate and unify a set of sub-models with well-accepted software paradigms of a system into a unified model represented in XML. We have verified the feasibility of the approach through a set of experiments. Three of these show how our model can overcome problems inherent in ripple effects.

In sum, our XUM approach improves software evolution and enhances software maintenance. XUM facilitates software evolution by exploiting the unification and the common points of models. Its unification links, which connect components of models in phases, enable systematic software reuse for analysis and design at the earliest stages in the software life cycle. XUM simplifies maintenance in all its aspects by upholding the connections of models and source codes of a system, while at the same time preserving the reset conditions of those in the views of the original paradigms and models. Users can update sub-models of a system from any modeling techniques or paradigms as needed. Any implicit inconsistencies will present themselves in the other related sub-models, allowing maintainers to deal with them in a clear, systematic manner. According to the discussion and validation presented above, we believe that the XUM approach proposed in this paper can be extended and benefit more activities in various processes for software evolution and, especially, for software maintenance.

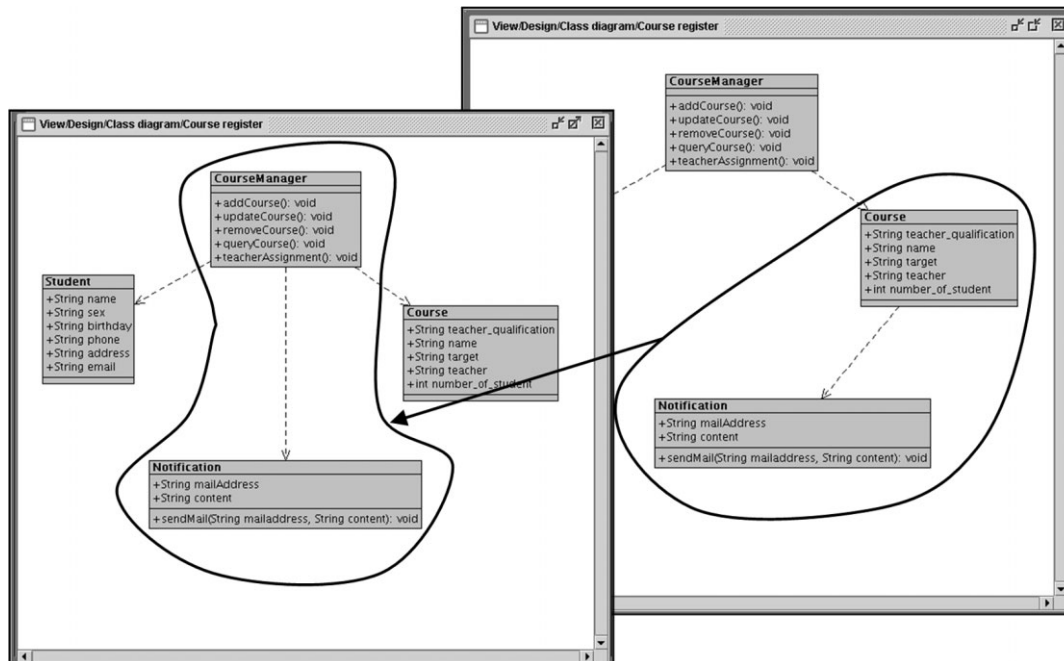


Figure 18. The correction of the change to the class diagram 'Course Register'.

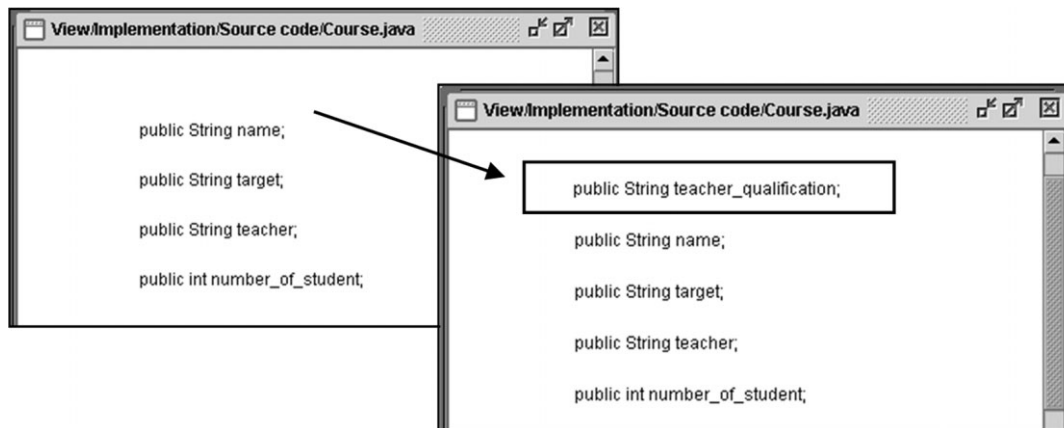


Figure 19. Updating the source code 'Course.java' by adding new attribute 'Teacher.Qualification'.

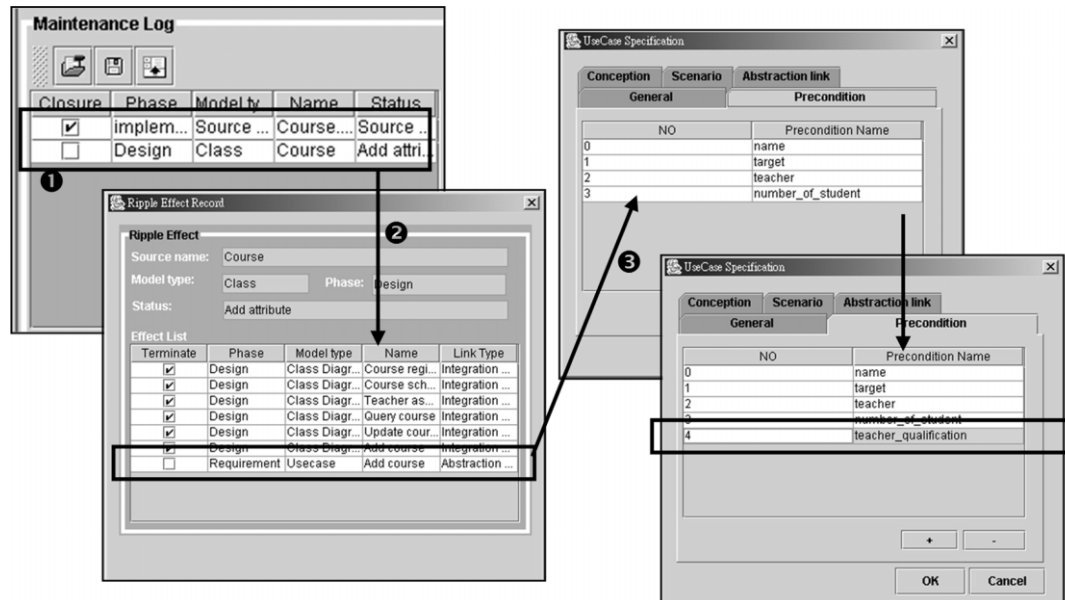


Figure 20. The update from a source-code change to the corresponding ripple effects in the other phases.

Before we close, we would like to point out six areas where more detailed studies of XUM application are needed.

The first of these areas is the linking of reuse technologies to the software processes—analysis, design, implementation, and maintenance—with a unified model. Applying reuse at an earlier phase of the software life cycle can reduce the software cost and increase software productivity greatly. Without integrating and unifying the models used in each phase, the links from the phases of requirement and design to implementation are missing. Therefore, to reuse software components in models in the early phases will be difficult without linking the models to their corresponding source codes. With the support of the unified model, the integration of software reuse technologies promises to be a fruitful direction for further study.

Second, further study comes in the modeling of design patterns. Most of the available design patterns are not yet formally specified and modeled. Although we have tried to capture the information of the design patterns in our model, they are not well specified yet. Formal modeling of design patterns is another important goal for the future. In particular, when a formal design pattern is applied together with other paradigms, some extra issues arise related to integration and unification. This entire area opens up avenues of additional research.

The third area of investigation includes the verification of consistency of the models themselves. Not only is the inter-model consistency important, but making the models in the software life cycle self-consistent is also a very crucial task. Some studies have worked in this direction; for



Table III. A detailed comparison of maintainer performance with the subject system.

Maintenance test cases	No. of models to be found and modified	Completeness (models found/ models should be found)	Ripple-coverage (no. of models that have to be checked out)	Time-cost (locating time + repair time) [unit: hours]
(1) M2M Using XUMIT	6	100% (6/6)	38 (22 + 8 + 3 + 3 + 1 + 1) (1.70% of 2232)	(0.4 + 2)
Traditional way		67% (4/6)	2232 (372 × 6)	(3.5 + 2)
(2) S2S Using XUMIT	4	100% (4/4)	27 (22 + 3 + 1 + 1) (1.81% of 1488)	(0.3 + 1.5)
Traditional way		75% (3/4)	1488 (372 × 4)	(1.5 + 1.5)
(3) P2P Using XUMIT	6	100% (6/6)	173 (157 + 1 + 8 + 3 + 3 + 1) (7.75% of 2232)	(0.7 + 2)
Traditional way		50% (3/6)	2232 (372 × 6)	(4.8 + 2)

example, Holzmann offered a verification system for models of distributed software systems [30]. Another example is that of Gunter *et al.*, who presented a reference model for requirements and specifications [31]. Our unified model helps the consistency checking between sub-models/views, yet the feature of verification in a sub-model/view needs to be worked out.

The fourth area for further study lies in XML-based program analysis. Currently, most source codes are kept in a plain-text format with no structural representation. As a result, both high-level information—such as requirement documents and design documents—and the comments written in source codes have no explicit links to the corresponding code segments. Representing programs in XUM that link source codes to the related elements/documents can facilitate the program analysis. Because XML documents are represented in DOM and because the compilers of XML are already available, efforts to implement analysis tool sets are much easier.

Fifth, future study is needed in the XUM-based software environment. A software environment that supports our XUM approach and provides more automation, guidance, distributed processing with remote data exchange, and assistance to the activities of software evolution and maintenance is another important future task.

Finally, in this paper, only parts of the UML-based paradigms have been explicitly specified and involved in the experiments, e.g., class diagrams in the phase of software analysis of the software maintenance life cycle. Involving more parts of the UML-based paradigms offers additional avenues of research and study.



These six possible future ‘works’ indicate where we believe the most important contributions to our field can be made—within the context of our proposed XUM approach. We plan to extend XUM and XUMM to embrace all the materials of modeling, design, implementation, and documentation. Further experiments for a comprehensive XUM environment and the tool sets are now being carried out to accomplish the goals of enhancement and unification of software evolution and software maintenance.

## APPENDIX. XUMM SCHEMA

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:complexType name="ComponentType">
    <xs:sequence>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="name" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="AssociationType">
    <xs:sequence>
      <xs:element name="link_id" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="to" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Unification_linkType"/>
  <xs:element name="actor">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="ComponentType"/>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="xumm">
    <xs:annotation>
      <xs:documentation>Comment describing your root element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="requirement">
          <xs:complexType>
            <xs:sequence>
              <xs:element ref="actor" minOccurs="0" maxOccurs="unbounded"/>
              <xs:element ref="usecase" minOccurs="0" maxOccurs="unbounded"/>
              <xs:element ref="usecase_diagram" minOccurs="0"
maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="design">
          <xs:complexType>
```



```
<xs:sequence>
  <xs:element ref="design_class" maxOccurs="unbounded"/>
  <xs:element ref="class_diagram" minOccurs="0"
maxOccurs="unbounded"/>
  <xs:element ref="collaboration_diagram" minOccurs="0"
maxOccurs="unbounded"/>
  <xs:element ref="design_pattern" minOccurs="0"
maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="implementation">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="framework"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="unification_link">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="integration_link" minOccurs="0"
maxOccurs="unbounded"/>
      <xs:element ref="abstraction_link" minOccurs="0"
maxOccurs="unbounded"/>
      <xs:element ref="sourcecode_link" minOccurs="0"
maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="usecase" type="ComponentType"/>
<xs:element name="usecase_diagram">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="usecase_association" type="AssociationType"
maxOccurs="unbounded"/>
      <xs:element name="relation_type" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="design_class">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="ComponentType">
        <xs:sequence>
          <xs:element name="attribute" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="id" type="xs:string"/>
                <xs:element name="name" type="xs:string"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
```



```

        <xs:element name="type" type="xs:string"/>
        <xs:element name="limit" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="operation" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="id" type="xs:string"/>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="return_type" type="xs:string"/>
        <xs:element name="limit" type="xs:string" minOccurs="0"/>
        <xs:element name="argument" minOccurs="0"
maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="id" type="xs:string"/>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="type" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="class_diagram">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="association" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="link" type="AssociationType"/>
            <xs:element name="type" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="collaboration_diagram">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="association" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="link" type="AssociationType"/>

```



```
        <xs:element name="sequence" type="xs:string"/>
        <xs:element name="message" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="design_pattern">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="pattern" type="ComponentType"/>
            <xs:element name="role" type="ComponentType" minOccurs="0"
maxOccurs="unbounded"/>
            <xs:element name="structure" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="link" type="AssociationType"/>
                        <xs:element name="type" type="xs:string"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="collaboration" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="link" type="AssociationType"/>
                        <xs:element name="sequence" type="xs:string"/>
                        <xs:element name="message" type="xs:string"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
            <xs:element name="constraint" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="role_from" type="xs:string"/>
                        <xs:element name="constraint_rule" type="xs:string"/>
                        <xs:element name="role_to" type="xs:string"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="framework">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="framework_id" type="xs:string"/>
            <xs:element name="framework_name" type="xs:string"/>
            <xs:element ref="sourcecode"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="sourcecode">
    <xs:complexType>
        <xs:sequence>
```



```

<xs:element name="package" type="ComponentType"/>
<xs:element name="class">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="class_id"/>
      <xs:element name="class_name"/>
      <xs:element name="attributes" minOccurs="0"
maxOccurs="unbounded">
        <xs:complexType/>
      </xs:element>
      <xs:element name="methods" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType/>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="integration_link" type="Unification_linkType"/>
<xs:element name="abstraction_link" type="Unification_linkType"/>
<xs:element name="sourcecode_link" type="Unification_linkType"/>
</xs:schema>

```

## ACKNOWLEDGEMENTS

We thank the referees for their valuable and extensive comments.

## REFERENCES

1. Sommerville I. *Software Engineering* (5th edn). Addison-Wesley: Reading MA, 1996; 4–19.
2. Bennett KH. An overview of maintenance and reverse engineering. *The REDO Compendium*. Wiley: Chichester UK, 1993; 3–18.
3. Canfora G, Cimitile A. Software maintenance. *Handbook of Software Engineering and Knowledge Engineering*, vol. 1. Knowledge Systems Institute: Skokie IL, 2001; 91–120.
4. Booch G. *Object-oriented Design with Applications*. Benjamin/Cummings: Redwood City CA, 1991; 1–25.
5. ISO. *Quality Systems—Model for Quality Assurance in Design/Development, Production, Installation, and Servicing, ISO9001* (rev. edn). International Organization for Standardization (ISO): Geneva Switzerland, 1994; 1–32.
6. ISO/IEC JTC1/SC7/WG10. *Software Process Assessment—Part 5: An Assessment Model and Indicator Guidance* (1st PDTR). International Organization for Standardization (ISO): Geneva Switzerland, 1996; 1–138.
7. Jenner MJ. *Software Quality Management and ISO 9001*. Wiley: Chichester UK, 1995; 47–160, 223–234.
8. Software Engineering Institute (SEI). *Software, Systems Engineering, and Product Development Capability Maturity Models® (CMMs®)*. Software Engineering Institute (SEI), Carnegie Mellon University: Pittsburgh PA, 2000, <http://www.sei.cmu.edu/cmm/cmms/transition.html> [27 January 2003].
9. Object Management Group. *OMG Unified Modeling Language Specification*, version 1.4. Object Management Group, Inc.: Needham MA, 2001, [http://www.omg.org/technology/documents/recent/omg\\_modeling.htm](http://www.omg.org/technology/documents/recent/omg_modeling.htm) [27 January 2003].
10. Anne CL. XML seen as integral to application integration. *IT Professional* 1999; 2(5):12–16.
11. Graham IS, Quin L. *XML Specification Guide*. Wiley: New York NY, 1999.
12. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley: Reading MA, 1995; 3–15.
13. Chu WC, Lu CW, Chang CH, Chung YC. Pattern based software re-engineering. *Handbook of Software Engineering and Knowledge Engineering*, vol. 1. Knowledge Systems Institute: Skokie IL, 2001; 767–786.



14. Holland I. The design and representation of object-oriented components. *Doctorial dissertation*, Northeastern University: Boston MA, 1993. Also available at PhD Theses (Northeastern University). <http://www.ccs.neu.edu/home/lieber/theses-index.html> [27 January 2003].
15. Johnson RE, Foote B. Designing reusable class. *Journal of Object-Oriented Programming* 1988; **1**(2):22–35.
16. Lano K, Malik N. Reengineering legacy applications using design patterns *Proceedings of the 8th International Workshop on Software Technology and Engineering Practice—1997*, IEEE Computer Society Press: Los Alamitos CA, 1997; 326–338.
17. Meyer B. Tools for the new culture: Lessons from the design the Eiffel libraries. *Communications of the ACM* 1990; **33**(9):68–88.
18. Ossher H, Kaplan M, Harrison W, Katz A, Kruskal V. Subject-oriented composition rules. *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference 1995*, special issue of SIGPLAN Notices. ACM Press: New York NY, 1995; 235–250.
19. Paul S, Prakash A. Framework for source code search using program patterns. *IEEE Transactions on Software Engineering* 1994; **22**(6):463–475.
20. Xiao C. Adaptive software: Automatic navigation through partially specified data structures. *Doctorial dissertation*, Northeastern University: Boston MA, 1994. Also available at PhD Theses (Northeastern University). <http://www.ccs.neu.edu/home/lieber/theses-index.html> [27 January 2003].
21. Moser S, Nierstrasz O. The effect of object-oriented frameworks on developer productivity. *IEEE Computer* 1996; **29**(9):45–51.
22. Chapin N. Software maintenance life cycle. *Proceedings Conference on Software Maintenance—1988*. IEEE Computer Society Press: Los Alamitos CA, 1988; 6–13.
23. Murphy GC, Notkin D, Sullivan KJ. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering* 2001; **27**(4):364–380.
24. Cheng BHC, Campbell LA, Wang EY. Enabling automated analysis through the formalization of object-oriented modeling diagrams. *Proceedings of the International Conference on Dependable Systems and Networks—2000 (DSN 2000)*. IEEE Computer Society Press: Los Alamitos CA, 2000; 305–314.
25. Fyson MJ, Boldyreff C. Using application understanding to support impact analysis. *Journal of Software Maintenance: Research and Practice* 1998; **10**(2):93–110.
26. Queille JP, Voidrot JF, Wilde N, Munro M. The impact analysis task in software maintenance: A model and a case study. *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1994; 234–242.
27. Yau SS, Colferro JS. Some stability measures for software maintenance. *IEEE Transactions on Software Engineering* 1980; **6**(6):545–552.
28. Deitel H, Deitel P, Nieto T, Lin T, Sadhu P. *XML How to Program*. Prentice Hall: Upper Saddle River NJ, 2001; 1–9.
29. Suzuki J, Yamamoto Y. Making UML models exchangeable over the internet with XML: UXF approach. *Proceedings of UML'98: Beyond the Notation—International Workshop*. Springer: Heidelberg Germany, 1998; 65–74.
30. Holzmann GJ. The model checker SPIN. *IEEE Transactions on Software Engineering* 1997; **23**(5):279–295.
31. Gunter CA, Gunter EL, Jackson M, Zave P. A reference model for requirements and specifications. *IEEE Software* 2000; **17**(3):37–43.

#### AUTHORS' BIOGRAPHIES



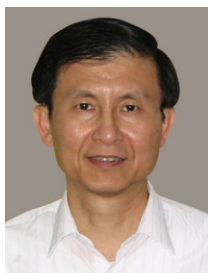
**Chih-Wei Lu** is currently an associate professor at the Department of Information Management, and also chief of the Computer Center, Hsiuping Institute of Technology, Taiwan. His research interests include component-based software engineering, design pattern, software reuse, and software maintenance. He received his BS degree in information science from Tunghai University in 1987, his MS degree in computer science from University of Southern California in 1992, and his PhD degree in information engineering and computer science from Feng Chia University in 2003.



**William Cheng-Chung Chu** is a professor and the chairman of the Department of Computer Science and Information Engineering at Tunghai University, Taiwan. From 1994 to 1998, he was an associate professor at the Department of Information Engineering and Computer Science at Feng Chia University. He was a research scientist at the Software Technology Center of the Lockheed Missiles and Space Company, Inc., where he received special contribution awards in both 1992 and 1993 and a PIP award in 1993. In 1992, he was also a visiting scholar at the Department of Engineering Economic Systems at Stanford University, where he was involved in projects related to intelligent knowledge-based expert systems. His current research interests include software engineering, embedded systems, and E-learning. Dr Chu received his MS and PhD degrees from Northwestern University in Evanston Illinois, in 1987 and 1989, respectively, both in computer science. He has edited several books and published over 100 referred papers and book chapters, as well as participating in many international activities, including organizing international conferences. Dr. Chu has received research awards and funded research grants.



**Chih-Hung Chang** received his BS and MS degrees in information engineering and computer science from Feng Chia University, Taichung, Taiwan, in 1995 and 1997, respectively. He is currently a PhD candidate in computer engineering at Feng Chia University, under the supervision of Dr. William C. Chu. His research interests include software maintenance, software re-engineering, design patterns, component-based software engineering, reuse, and software process document formalization.



**Don-Lin Yang** is currently a professor and the chair at the Department of Information Engineering and Computer Science at Feng Chia University. Prior to joining the university in 1991, he was a staff programmer at IBM Santa Teresa Laboratory from 1985 to 1987 and a member of the technical staff at AT&T Bell Laboratories from 1987 to 1991. His research interests include software engineering, distributed and parallel computing, and data mining. He is a member of the IEEE computer society and the ACM. He received his BE degree in computer science from Feng Chia University in 1973, his MS degree in applied science from the College of William and Mary in 1979, and his PhD degree in computer science from the University of Virginia in 1985.

**Wen-Da Lian** received his BS and MS degrees in computer science and information engineering from Tunghai University, Taichung, Taiwan, in 2000 and 2002, respectively. His research interests include software reuse, software model and API design, design pattern, and software maintenance.