COMO: A UML-Based Component Development Methodology

Sang Duck Lee, Young Jong Yang.
Realtime Computing
ETRI Computer & Software Technology Laboratory
Daejon, Korea 305-333
yjyang@seri.re.kr

Eun Sook Cho, Soo Dong Kim, Sung Yul Rhew
Dept. of Computer Science
Soongsil University
Seoul, Korea 156-743
sdkim@computing.soongsil.ac.kr

Abstract
Many organizations have begun to consider implementing applications using reusable components. They envision that writing a component once and reusing it in many subsequent software systems will effectively amortize the development cost among all users. Recent technology advances such as JavaBeans/EJB, COM, CORBA, and others spur this interest. However, systematic development process and practical instructions for building software components have not been studied enough. In this paper, we propose a practical object-oriented component development methodology (COMO) that can be utilized in developing software components. COMO extends Unified Modeling Language (UML) and Rational's Unified process with semantics related to component development. Since COMO provides systematic process and comprehend instruction for each development task, we believe it can be effectively used in modeling and implementing software components.

1. Introduction
Component-based software engineering is generating tremendous interest not just in the software community but also in numerous industry sectors. Recent technology advances such as JavaBeans/Des97, COM/COM+/Reg97, EJB/Sun97, CORBA, and others spur this interest/Bro98. Currently, these component architectures for component development are being introduced, but systematic development process and detailed instructions for building OO components have not been studied enough. In addition, it has not been clearly defined how to apply OMG's Unified Modeling Language (UML) in building OO components.

In this paper, we propose a practical OO component development process and modeling techniques that can be adaptable to practical component development. The proposed modeling techniques are based on UML notations.

Section 2 of this paper discusses existing component development methodologies such as Catalysis and SCIPLO. In the Section 3, our component modeling techniques and development process are described. Section 4 presents a case study and the assessment of the proposed modeling techniques applied to Electronic Commerce (EC) domain. Finally, we describe concluding remarks in Section 5.

2. Related Work

2.1. Catalysis Methodology
Catalysis is a rapidly emerging next generation methodology that provides support for component-based development with objects and frameworks [Sou98]. Catalysis development process consists of analysis, design, implementation, and testing phases. Spiral model is applied into this development process. Analysis, design, implementation, and testing phases are iteratively processed, and total system is incrementally developed.

Catalysis method provides various notations and guidelines for component and application development. It provides pattern processes to adapt development process into various projects. It provides type models for component representation.

2.1.1. Limitations of Catalysis
Flows between tasks are not concretely defined. Therefore, we are hard to track or control tasks during component development. Because input or output artifacts
for each task are roughly provided, we cannot synchronize artifacts. Concrete instructions for carrying out task are not provided. Also, concrete guidelines for component identification are not provided. Message flows between classes in a component are not described. It does not provide customization parts (or hot spots) for each component.

2.2. SCIPIO Method

SCIPIO contains business process modeling, workflow management and object-oriented analysis and design[Ver98]. This methodology focuses on component-based software development rather than component development. Therefore, this full life cycle includes some activities related with component development. Application developer defines workflows through business process analysis and design, and identifies reusable components through analysis of existing applications. And then, they develop new applications.

2.2.1. Limitations of SCIPIO

It does not provide concrete flows between phases or tasks; How to the output of a task is reflected by the subsequent task. Also, it does not provide concrete instructions for each task in a phase. Guidelines related with component development are not enough described. It does not provide how to identify components. Also, there are concrete instructions for component design; how the component interfaces are defined and customization parts are described in a component.

3. Component Modeling Technique(COMO)

3.1. Modeling Criteria

In this paper, we consider modeling criteria of our modeling techniques. First, we only focus on component itself development. We consider the practicality of component development process. We propose workflow between tasks, and define concrete modeling guidelines. Second, we consider effectiveness of component modeling. Current component diagram of UML presents component information roughly. There is no message flows or classes within a component. Therefore, we extend current UML notations to model component through adding message flows and classes into component diagram. Third, we consider consistency of diagrams. Artifacts created in previous tasks may be used in next tasks. Also, we propose cross checking in modeling process. For example, in case of component sequence diagram, message flows between components are mapped into interfaces of component diagram. Fourth, we consider reliability of component modeling.

3.2. Component Development Process

We define a process for component development based-on Unified process. Component development process consists of four phases (see Figure 1). During the component development process, we iteratively and incrementally develop a complete component. This implies describing the component requirements, fleshing out the component design, and completing the component build and test of component.

Figure 1. Component Development Process

3.3. Domain Analysis Phase

The goals of domain analysis phase are to understand the problem domain and identify component requirements. Especially, because business component should be made in a unit that many applications may commonly reuse, domain analysis is very important. Workflows between tasks in this phase are shown in Figure 2.

Figure 2. Domain Analysis Phase

3.3.1. Identifying Domain Requirement Set

When we identify domain requirement set, we focus on functional requirements from many application requirement specifications. The reason is that component is focused on common functions among applications.

However, there are different term representations in the identified domain requirement set. Therefore, after defining term dictionary which contains standard term description, we should refine domain requirement set according to term representations in term dictionary. Also, we define the function dictionary containing standard
function descriptions in specific domain. This function dictionary is used for use case modeling and component identification.

3.3.2. Existing Commonality

In order to extract common functional requirements, we adopt use case modeling technique into domain requirement set. We identify overlapped functions among applications, and define overlapped functions as common functions. For the identified common functions, we make use case diagram. A technique of use case modeling is based on UML’s technique [Rat97]. We identify use cases by grouping cohesive functions from identified common functions list. Cohesive functions are identified through grouping functions that have high similarity functionally. Additionally, we identify cohesive functions by grouping functions that manipulate the same attributes.

We extend UML’s use case description format by adding “Functions Included” contents. Additionally, we note variant parts by “(variant attributes)” notation. For example, in case of ‘Order Product’ use case, we may note payment attribute as (credit card, online cash) form. We describe cohesive function described in previous step in “Functions Included” section.

3.3.3. Identifying Variability

In order to model flexible component, we identify variant attribute, logic, and workflow that are different according to applications in specific domain.

In case of attributes, we identify attributes used in common use case per application, and identify variant attributes according to application.

![Structured English Use Case Description Format](image)

Figure 3. Structured English Use Case Description Format

In case of logic, we identify operations included in each use case description. Also, in case of workflow, we identify different sequences of operations in use case description according to application use case. However, because use case description is described textually, it is hard to identify operations. Therefore, we transform use case description into Structured English format. In this format, notation ‘A’ means alternative flow, ‘E’ means exception flow, and ‘V’ means variant flow (See Figure 3).

After identifying variability, for each variability, we determine default value, type, determination value. Identified variability information such as variability number, name, type, default value, and determination value is listed in variability list (See Table 1).

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default Value</th>
<th>Determination</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcessPayment</td>
<td>Logic</td>
<td>Card</td>
<td>NonPredictable</td>
</tr>
</tbody>
</table>

3.3.4. Constructing Conceptual Object Model

In the previous task, we only focus on homogeneous contents functionally for target domain. However, we should identify major business concepts in a target domain to model static aspect of target domain.

In this task, we identify business concepts from domain requirement set, term dictionary, and domain knowledge. In order to identify business concepts, we apply class identification techniques of OMT [Rum91]. In a conceptual class diagram, we only represent class, attributes, relationships, and operations in class diagram.

3.3.5. Identifying Components

We propose two clustering techniques: use case clustering technique and use case class clustering technique. Especially, we propose a clustering algorithm for use case and class clustering.

Firstly, use case clustering means that we should cluster cohesive use cases by considering <+extends> relationship between use cases. We cluster use cases related with <+extends> relationship. Secondly, we cluster cohesive use cases and classes into component by applying proposed clustering algorithm.

![Use Case/Class Matrix](image)

Table 2. Use Case/Class Matrix

<table>
<thead>
<tr>
<th>Class 1</th>
<th>Class 2</th>
<th>Class 3</th>
<th>Class 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case 1</td>
<td>C</td>
<td>R</td>
<td>C</td>
</tr>
<tr>
<td>Use Case 2</td>
<td>W</td>
<td>R</td>
<td>C</td>
</tr>
<tr>
<td>Use Case 3</td>
<td>R</td>
<td>C</td>
<td>R</td>
</tr>
</tbody>
</table>

Before the clustering use cases and classes, we define matrix for use cases and classes. When we make out matrix, we assign value according the relationship between use case and class. In this thesis, we define the relationship between use case and class as four types: “Create”, “Delete”, “Write”, and “Read”. Also, we give priority for relationship as following: “Create>Write/Delete>Write” (See Table 2).
We cluster use cases and classes by applying clustering algorithm illustrated in Figure 4.

```
if (B \in C) \land (D \in C) \land (E \in C) \land (F \in C)

As illustrated in Figure 6, component specification consists of component name, brief description, workflow, contained classes, cross-references, friend component, commonality, and variability. If a component contains several use cases, workflow is described for each use case. We add classes into contained classes section. We describe referencing use cases in cross-references section, and assign related components which relate dependency with itself. In the commonality section, commonality and variability table defined in the previous tasks is assigned. In the variability section, variability list is illustrated.

Figure 6. Component Requirement Specification

3.3.8. Refining Domain Analysis Model

The purpose of this task is to review and refine artifacts of domain analysis phase. In this task, we should check consistency among artifacts or completeness of each artifact.

3.4. Component Design Phase

Component design phase consists of six tasks, and these tasks are processed iteratively (See Figure 7).

Figure 7. Component Design Phase

3.4.1. Identifying Message Flows

We make out sequence diagram for each use case in a component, and represent message flows between objects or message flows between object and component. In this paper, we extend sequence diagram. We add component instance in sequence diagram, and represent message flows between object and component(See Figure 8).

Figure 8. Component Design Phase (Sequence Diagram)
These message flows are extracted from use case description for each component. Each event is mapped into message flows between objects in a component.

3.4.2. Defining Class Interface

Elaborating conceptual class diagram means adding operations in classes included in conceptual class diagram and adding additional objects related with component implementation. In order to add operations in classes, we refer object sequence diagram resulted in previous task. If one object receives message from the other objects, we should map the message into methods of the object. Because one component consists of one or more objects, message flows between objects should be included. These message flows are reflected on the component interface.

3.4.3. Defining Customization Policy

We classify defining customization policy into three customization policies: customization policy for variant attributes, customization policy for variant logic, and customization policy for variant workflow.

Defining customization policy for variant attributes is classified into two types; one is determining customization policy for variant values of attributes, and the other is determining customization policy for variant attribute set having fixed operations. We define methods of variant attributes as set<AttributeName> pattern. And those set methods are assigned into corresponding class. Otherwise, in order to determine customization policy for variant attribute set having fixed operations, we define variant attributes set as template class.

3.4.5. Defining Component Specification

In order to define required interface, we select sending message flows from the controller to object of other component or select sending message flows from the object in a component to object of other component. We define such interface of other component as a required interface of target component as depicted in Figure 11.
First, we define component contract. Component contract provides what component users need to use components. Also, it provides what component developer needs to implement interfaces of component. In order to define component contract, we should define pre-condition, post-condition, and exceptions for each method of component interface in a component.

In this paper, we propose component contract as extended IDL of CORBA component model because IDL of CORBA is independent on specific programming languages (see Figure 12).

![Component Contract](image)

**Figure 12. Component Contract**

In IDL form, component is described by "Component" syntax, and several interfaces are contained in a component. Also, interfaces of component are classified into provided interfaces and required interfaces. Provided interfaces are described by "Provides" syntax, and required interfaces are described by "Requires" syntax. Especially, when we describe required interfaces, we add component name through scope operator. Additionally, pre-condition, post-condition, and exception are described for each operation in an interface of component.

![Component Specification](image)

**Figure 13 Component Specification**

After defining component contracts, we make out component specification. As illustrated in Figure 13, component specification consists of component name, brief description, participants, static model, dynamic model, and component contract sections. In the participants section, class name contained in a component is described. Component diagram is represented in the static model section, sequence diagrams are described in a dynamic model section, and component contract is illustrated in the component contract section.

### 3.4.6. Refining Design Model

The purpose of this task is to review artifacts of component design phase. We should check consistency among artifacts and synchronize artifacts by consistency checking.

### 4. Case Study and Assessment

In this section, we propose a case study for EC (Electronic Commerce) domain applied with proposed framework development process and describe assessment between proposed process and existing development process.

#### 4.1. Case Study

Currently, many EC sites are increasing in Internet. Therefore, if we develop reusable components for the domain, more EC applications will be developed with short and efficient. Also, because there are many reusable UI components as well as reusable business logic such as order, payment, and delivery, we considered this domain as appropriate domain for component development.

##### 4.1.1. Domain Analysis Phase

In order to common functional requirements, we gathered existing EC application requirement specifications. And we defined term dictionary and function dictionary based on domain requirement set. After defining function dictionary for electronic commerce domain, we refined functional requirements of each application requirement specification based on function dictionary.

![Use case diagram for EC](image)

**Figure 14. Use case diagram for EC**
In the DA2 task, we extracted common requirements from normalized functions described in domain requirement set. In the electronic commerce domain, we identified four actors (e.g. Customer, Administrator, Supplier). Identified use cases are shown in Figure 14.

In the DA3 task, we identified five classes (e.g. 'Order', 'Customer', 'Product', 'Payment', 'ShoppingCart'). In the DA4 task, we identified three components such as 'Customer Management', 'Order Management', and 'Product Management'.

![Figure 15. Allocated business objects](image)

In the DA5 task, we allocated business objects modeled in the previous task identified components. Additionally, dependency relationships are described among 'Order Management' component, 'Customer Management', and 'Product Management' components because 'Order Management' component refers 'customer' class of 'Customer Management' component and 'product' class of 'Product Management' component. The result of allocated business objects is depicted in Figure 15. In the DA6 task, we defined component requirement specification for each component.

### 4.1.2. Component Design Phase

For the first, we identified message flows per each use case based on use case description for each use case. We represented message flows in sequence diagram.

![Figure 16 Component interface of 'Order Product']

In the CD2 task, we defined methods of classes based on identified message flows. In the CD3 task, we defined customization methods for variant attributes and variant methods. For example, we defined customization method for 'Job' variant attribute as 'setJob(Job job)'. Also, we defined customization method for 'processPayment()' variant behavior as 'processPayment(Payment payment)'.

In the CD4 task, we identified component interface for each component, and identified methods of each component interface through sequence diagram. An example of defining component interface is illustrated in Figure 16. As shown in Figure 16, dashed rectangle means methods of required interface in a component, and rectangle boxes means methods of provided interface in a component. In this case study, we identified six component interfaces, and defined public methods in each identified component interface.

![Figure 17 Component Diagram for EC](image)

After defining component interfaces, we constructed component diagram as shown in Figure 17.

### 4.1.3. Defining Component Specification

In this task, we defined component specifications for each component. In the previous of defining specifications, we defined component contracts for each component. An example of component contract for 'Order Management' component is illustrated in Figure 18.

![Figure 18. Component Contract for 'Order Management' Component](image)
4.2. Lessons Learned

We developed business components of EC components based on EJB technology. During component development, we applied our modeling techniques under component development process. Systematic development process allows us to develop components in short period, and produces high-quality component. We found that 80% of application code was reused by component. Therefore, the size of application code is considerably reduced.

Also, we compare proposed modeling techniques with other modeling techniques such as Catalysis and SCIPIO. Comparing modeling techniques, we propose characteristics of our modeling techniques. We compare modeling techniques based on modeling criteria proposed in Chapter 3 (See Table 4).

5. Concluding Remarks

In this paper, we proposed a practical component development methodology that extends UML and Unified process. We defined development phases and tasks for each phase. We proposed essential modeling techniques for component development and applied UML notations to each task. We applied COMO in developing components for electronic commerce application to assess its practical applicability and effectiveness. Through the application on a commercial project, we believe that components can be more efficiently developed through COMO and high quality components can be produced.

### Table 4. Comparison of Modeling Techniques

<table>
<thead>
<tr>
<th>Modeling Criteria</th>
<th>Catalysis</th>
<th>SCIPIO</th>
<th>COMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development process is defined?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Task flows are defined?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Concrete modeling guidelines are proposed?</td>
<td>Partially</td>
<td>Partially</td>
<td>Yes</td>
</tr>
<tr>
<td>Customizable parts are represented?</td>
<td>Partially</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Practicality</td>
<td>Hard</td>
<td>Hard</td>
<td>Easy</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>Medium</td>
<td>Medium</td>
<td>High</td>
</tr>
</tbody>
</table>

[References]


