A DOMAIN ANALYSIS AND MODELING METHODOLOGY FOR COMPONENT DEVELOPMENT*

EUN SOOK CHO\textsuperscript{1}, SOO DONG KIM\textsuperscript{1,5} and SUNG YUL RHEW\textsuperscript{1,5}

\textsuperscript{1}College of Computer and Information Science, Dongduk Women’s University 23-1 Wolgok-dong, Sangbuk-ku, Seoul 136-714, Korea
echo@dongduk.ac.kr

\textsuperscript{2}Department of Computer Science, Soongsil University, 1-1 Sangdo-5 dong, Dongjak-gu, Seoul 156-743, Korea
sdkim@comp.ssu.ac.kr
syrhew@comp.ssu.ac.kr

Submitted 5 December 2002
Revised 19 June 2003
Accepted 26 June 2003

Component-based development (CBD) has been widely accepted as an effective paradigm for building software with reusable components, consequently reducing efforts and shortening time-to-market. The components used in CBD should provide a domain standard or common functionality so that they can be shared and reused by family members in a domain. Moreover, variability among family members should also be modeled in components so that family member-specific business logic can be realized through component customization. Therefore, the degree of commonality and customizability determines the range of component applicability in CBD. In this paper, we propose a practical COmponent MOdeling methodology (COMO) for analyzing domain requirements and designing highly reusable components. COMO extends Unified Modeling Language and Rational Unified Process with notations and semantics relevant to CBD. For each activity of the process, comprehensive instructions and artifact templates are provided so that software components can be modeling effectively. We also provide a comprehensive case study of applying the methodology in building an electronic commerce application.

Keywords: Component-based development; domain analysis; commonality and variability.

1. Introduction

As one of the promising software reuse disciplines, component-based development (CBD) has been widely accepted in both academia and industry as an effective paradigm for building software with reusable components [1], consequently reducing

*This work was supported by the Soongsil University Research Fund.
software costs and shortening time-to-market. As a larger-grained reuse unit that
objects, a component typically consists of several related classes/objects to provide
a system-level cohesive functionality.

Components are mainly developed for inter-organizational reuse whereas objects
are for intra-organizational reuse. Hence, components should provide a domain
standard or common functionality so that they can be shared and reused by many
family members in a domain. Moreover, variability among family members should
also be modeled in components so that family member-specific business logic can be
realized through component customization. Therefore, the degree of commonality
and variability (C&V) support in components determines the range of component
applicability, and domain analysis is a common and key activity for modeling C&V.

Most current component reference models such as CORBA Component Model
[2], Enterprise JavaBeans [3], and .NET [4, 5] extend object-oriented paradigm with
CBD related concepts and constructs. However, object-oriented methods do not
provide sufficient mechanisms for modeling components. Earlier component-based
methods support notion of components, interface design and component realization.

In this paper, we propose a practical COmponent MOdelling methodology
(COMO) for domain analysis and highly reusable component design. COMO ex-
tends Unified Modeling Language and Rational Unified Process with additional
notations and semantics relevant to CBD. It captures key concepts found in ear-
er CBD methods, but adds a systematic process with traceability and industry-
strength activity specifications. For each activity of COMO process, comprehensive
instructions and artifact templates are provided so that software components can
be effectively modeling. We also provide a comprehensive case study of applying
the methodology in building an electronic commerce application.

2. Related Work

Since the mid 90’s, several CBD methods have been introduced but relatively few
methods have been applied to industry projects. The CBD96 [6] methodology of
Sterling software does not define the development process. It only gives a component
reference model to support CBD development, but is not a complete development
methodology: there is no detailed design, coding, or testing tasks in this methodol-
gy. CBD96 uses new terminology or notations similar to OO concepts and UML
notations. Using these terminologies and notations can cause confusion about the
concepts.

Catalysis [7,8] is a rapidly emerging next-generation methodology that provides
support for component-based development with objects and frameworks. However,
it has some limitations. First of all, it is difficult to understand the models, and its
deliverables can be complex because it provides too many notations. Second, the
sequences of tasks in each phase are not well defined. How to apply the artifacts of
a previous task into the subsequent task is not clearly defined. Finally, beginners
may have difficulties applying this methodology into practical CBD development,
due to its high formalism.
The UNIFACE development methodology uses the industry-standard object-oriented modeling techniques and notations incorporated in the UML [9, 10]. However, like Catalysis, UNIFACE also provides no concrete instructions executed in each task. Therefore, it is hard for developers to apply this methodology in a practical project. Second, the tasks for component modeling, such as customization techniques or component interface definitions are not defined enough in the methodology. Third, there is a task of UNIFACE mapping in this methodology, but the task depends on a specific UNIFACE development environment. Therefore, this task is not necessary for developers who do not use UNIFACE tools.

However, these methodologies focus on component-based software development rather than component development. There are few concrete modeling instructions related to component modeling.

### 3. The Overall Process

The overall process of COMO consists of four phases as shown in Fig. 1: Domain Analysis, Preliminary Design, Detailed Design, and Implementation. However, the scope of this paper is limited to the first two phases, during which platform independent components are modeled.

The main goal of Domain Analysis phase is to analyze the target domain in order to derive C&V and to identify conceptual components. This phase consists of eight
activities, and each activity produces development artifacts which are then used by later activities. Figure 2 shows the eight activities of Domain Analysis phase and artifacts used by and generated by each activity. For example, the activity DA1 uses *Application Requirement Specifications* and produces *Term Dictionary* and *Function Dictionary* which are in turn used by the activity DA4.

The main goal of the second phase, *Preliminary Design*, is to design platform
Table 1. List of artifacts.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Artifacts</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DA1</td>
<td>Domain Requirement Set</td>
<td>A set of application requirement specifications in a domain, collected from existing applications and/or family members</td>
</tr>
<tr>
<td></td>
<td>Term Dictionary</td>
<td>A glossary of common and standard terms</td>
</tr>
<tr>
<td></td>
<td>Function Dictionary</td>
<td>Description of all the functions found in a domain</td>
</tr>
<tr>
<td>DA2</td>
<td>Common Functional Requirement Set</td>
<td>A requirement of functions which are common among family members</td>
</tr>
<tr>
<td></td>
<td>Use Case Diagram</td>
<td>A functional model using UML</td>
</tr>
<tr>
<td></td>
<td>Use Case Description</td>
<td>Textual and detailed description of use cases</td>
</tr>
<tr>
<td>DA3</td>
<td>Structured Use Case Description</td>
<td>Structured form of use case description, used to effectively identify event flows</td>
</tr>
<tr>
<td></td>
<td>Commonality and Variability Table</td>
<td>Table of common and variable features</td>
</tr>
<tr>
<td></td>
<td>Variability List</td>
<td>Description of variability in terms of variability name, type, scope, default value, and determination value</td>
</tr>
<tr>
<td>DA4</td>
<td>Conceptual Class Diagram</td>
<td>High-level view of class diagram</td>
</tr>
<tr>
<td>DA5</td>
<td>List of Components</td>
<td>A list of initially identified components</td>
</tr>
<tr>
<td>DA6</td>
<td>List of Refined Components</td>
<td>A list of refined components</td>
</tr>
<tr>
<td>DA7</td>
<td>Component Requirement Spec.</td>
<td>Specification of components, including name, functionality, workflows, classes, references, friend component and C &amp; V info</td>
</tr>
<tr>
<td>PD1</td>
<td>Sequence Diagram</td>
<td>Diagram describing interactions between objects and/or components</td>
</tr>
<tr>
<td>PD2</td>
<td>Specification Class Diagram</td>
<td>Description of specification level classes</td>
</tr>
<tr>
<td>PD3</td>
<td>Refined Class Diagram</td>
<td>Description of refined classes, including customization instruction for variable attributes, logics, and workflows</td>
</tr>
<tr>
<td>PD4</td>
<td>Component Diagram</td>
<td>Diagram describing components, their dependency and interfaces</td>
</tr>
<tr>
<td>PD5</td>
<td>Component Contract</td>
<td>Specifications of contract between components and interfaces</td>
</tr>
<tr>
<td></td>
<td>Component Specification</td>
<td>Detailed specification of components, including internal structure, dynamic model and component contract</td>
</tr>
</tbody>
</table>

We now summarize the artifacts for the two phases in Table 1. Table 1 shows the artifact’s name and description for each activity. Since the traceability among artifacts and seamless transition between artifacts are important criteria in COMO, a large number of intermediate artifacts are used.

independent components and their interfaces. This phase consists of six activities as shown in Fig. 3.
Table 2. Term dictionary.

<table>
<thead>
<tr>
<th>Standard Term</th>
<th>Meaning</th>
<th>Category</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>A client who shops products from internet shopping malls.</td>
<td>Entity</td>
<td>Member, User</td>
</tr>
<tr>
<td>Shopping Cart</td>
<td>A conceptual cart containing products selected by customer</td>
<td>Entity</td>
<td>Cart, Shopping Basket</td>
</tr>
</tbody>
</table>

Table 3. Function dictionary.

<table>
<thead>
<tr>
<th>ID</th>
<th>Function Name</th>
<th>Description</th>
<th>Functional Category</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Add Member</td>
<td>It verifies the information of new customer, and creates a customer ID.</td>
<td>Customer Management</td>
<td>Name, SSN, Birth date, Sex, Address, Phone, ...</td>
</tr>
<tr>
<td>F2</td>
<td>Search Member</td>
<td>It retrieves information of the customer using customer ID or other information.</td>
<td>Customer Management</td>
<td>Customer ID, Customer's other information such as ...</td>
</tr>
</tbody>
</table>

4. Domain Analysis (DA) Phase

In this phase, the target domain is analyzed to derive C&V information and to model preliminary components. Each activity of this phase is specified with instructions, artifact templates and examples below.

4.1. Identifying domain requirement set (DA1)

Since components are mainly for inter-organizational reuse, a comprehensive set of requirement specifications should be acquired from family members in order to determine C&V. If the set of requirement specifications is incomplete, virtual requirement specifications can be defined and added to this set with careful judgments to make the set more complete.

One of the common problems in comparing multiple requirement specifications is the heterogeneity of terms used. A single term may be used with different meanings, and multiple terms may have the same meaning. Hence, we need to normalize these terms before we identify the commonality. The Term Dictionary in Table 2 is used to define the standard terms used in the domain.

In the table, we specify standard terms in the first column, and describe the standard meaning in the next column. The category is the type of term, which can be an attribute, entity, function, etc. All aliases of a standard term are also entered in the table. Using the term dictionary, we normalize all the requirement specifications and now identify the functional requirement in the domain. Even with standard terms, there may often be a problem with heterogeneous descriptions of functions among family members. Hence, we use Function Dictionary as in Table 3,
which describes the standard meaning of each function and attributes used. This table is later used as input to use-case modeling.

4.2. Extracting commonality (DA2)

The commonality in CBD is the set of functionality or features which are standard or commonly used among family members [11]. The activity Extracting Commonality is to identify such commonality in a domain. By using the Function Dictionary, we can systematically compare multiple requirement specifications as in Fig. 4, where four requirement specifications, ARS#1 through ARS#4, are compared. In this example, a large number of functions such as F1 and F2 are common among the four members, and functions like F19 are common only among two members. If the number of family members is large, then a table is more suitable than diagram to compare the functions among members.

We now map the set of common functions to the units of UML use cases [12] since current practical component reference models are all based on object-oriented paradigm where use cases provide the functional aspect of the requirement. A function specified in a requirement specification can be in any granularity, and hence we need to consider the scope of each function in this step. The following tips may be used:

(1) A set of closely related functions should be grouped into a use case to provide cohesive system-level functionality. The degree of close relationship between functions is determined based on functional dependency.
Use Case Description #11

Name: Order Product

Brief Description: This is an use case that adds products into shopping cart, receives the information of payment and delivery of customer, and place an order.

Functions included: F12, F27, F30, F31

Main Flow
1. A customer requests ‘Order Product’.
2. The system displays order information to customer, and requests {payment} information, {delivery} information, and password.
3. A customer inputs {payment}, {delivery}, and password information (A1, A2).
4. After the system verifies customer’s input information (E1, E2, E3), calculate delivery charge (A3), and computes total amount.
5. The system processes payment (A4).
6. If processing payment is completed, the system adds customer’s order information, and decreases amounts of product.
7. The system displays order result.

Alternative Flows
A1.1: Customer Delivery – The system receives customer’s information.
A1.2: Other Delivery – The system requests customer to input delivery information.
A2.1: CreditCard Payment – The system requests customer to input card number, expiration date, and installment months. After verifying customer’s input information, it processes card payment.
A2.2: OnLine Payment – The system requests customer to input bank name, account number, owner, and deposit date. After verifying customer’s input information, it processes cash payment.
A3.1 AirlineDeliveryFee – The system computes delivery fee by adding airline fee of delivery location into basic fee.
A3.2 ShipmentDeliveryFee – The system computes delivery fee by adding shipment fee of delivery location into basic fee.
A3.3 LandDeliveryFee – The system computes delivery fee by adding landing fee of delivery location into basic fee.
A4.1 Processing CreditCard – The system sends card number to card company. If the acceptance number is received, the system processes credit card payment.
A4.2 Processing OnLine Cash – The system verifies account. If total amounts is deposited in the account, the system processes online cash payment.

Exception Flows
E1: Incorrect password is entered. The system displays “Password is incorrect” message, and requests password again or exits task.
E2: Incorrect payment is entered. The system displays “Payment is incorrect” message, and requests payment information again.
E3: Incorrect delivery is entered. The system displays “Delivery is incorrect” message, and requests delivery information again.

Fig. 5. An example of use case Description.
A component provides certain functionality and it holds data as objects. Hence, both

4.5. A component consists of multiple objects/classes, and hence we need to build an

4.4. use case model and object model should be used to identify candidate components.

Fig. 6. The classes identified in this activity are used in later activities in designing

variants, and a default value is used if the variability is not tailored by component

consumers. If a variation point has a closed set of known variants, it is marked

as `Predictable', otherwise `Non-predictable' indicates that the scope is open for

future additions.

use case description artifacts. A use case description includes a set of supported
functions and a function description lists attributes manipulated by the function.
By entering the attributes used by each use case for each family member in Table 4,
the common and variable attributes are identified mechanically.

Similarly, logic variability is identified by using use case description and function
description. Functions supported by each use case are listed for each family mem-
ber. When comparing functions, the semantics of functions should be considered
in addition to syntactic aspects such as function signatures. We use a naming con-
vention for functions with variability as the form of (function name) V (sequential
number). For example, if a function F4 has a variation point of three variants, we
express the logic variability as F4v1, F4v2, and F4v3.

Workflow variability is identified by examining use case descriptions which reveal
method invocation sequences of family members. Then, we enter the sequences in
the table to identify the C&V on workflows.

The next step after identifying variation points, we determine the scope of vari-
ation point using Variability List in Table 5. For each variation point, we enter its
variability type, scope, default value and determination. A scope is a set of possible
variants, and a default value is used if the variability is not tailored by component
consumers. If a variation point has a closed set of known variants, it is marked
as `Predictable', otherwise `Non-predictable' indicates that the scope is open for
future additions.

<table>
<thead>
<tr>
<th>Number</th>
<th>Variability Name</th>
<th>Variability Type</th>
<th>Scope</th>
<th>Default Value</th>
<th>Determination</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>processPayment( )</td>
<td>Logic</td>
<td>{Credit Card, On-line Cash}</td>
<td>Credit Card</td>
<td>Non-Predictable</td>
</tr>
<tr>
<td>V2</td>
<td>computeDeliveryFee()</td>
<td>Logic</td>
<td>{Ship, Airline, Land}</td>
<td>Land</td>
<td>Non-Predictable</td>
</tr>
<tr>
<td>V3</td>
<td>Credit Card</td>
<td>Attribute</td>
<td>{Visa, BC, Master}</td>
<td>None</td>
<td>Predictable</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
4.4. Constructing conceptual object model (DA4)

A component consists of multiple objects/classes, and hence we need to build an object model for the common requirement specification. Any object modeling techniques such as OMT [14] can be used to construct a conceptual class diagram as in Fig. 6. The classes identified in this activity are used in later activities to design components.

4.5. Identifying Components (DA5)

A component provides certain functionality and it holds data as objects. Hence, both use case model and object model should be used to identify candidate components. We use two-step clustering algorithm for component identification. The first step is to cluster use cases by considering their relationship. A use case depends on another use case if one requires the invocation of the other. For example, two use cases with \langle extends \rangle relationship should be clustered as a group.

The next step is to cluster use cases and their related classes into components by using Use Case/Class matrix shown in Table 6 and a clustering algorithm as shown in Fig. 8. Initially, we create the Use Case/Class matrix by considering their Create(C), Read(R), Write (W), and Delete (D) relationships. This table shows what classes are used by each use case and which of the four C, R, W or D relationship is dominantly used.
Once the matrix is constructed, we now cluster use cases and classes together into components by applying the clustering algorithm in Fig. 7. This algorithm mainly moves use cases, i.e. rows, to group closely related use cases and classes in
a single region in the matrix. The algorithm considers the four C, R, W, and D relationships. For example, the ‘C’ relationship is considered to be stronger than the ‘R’ relationship.

<table>
<thead>
<tr>
<th></th>
<th>Customer</th>
<th>Product</th>
<th>Shopping Cart</th>
<th>Order</th>
<th>Payment</th>
<th>Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join Membership</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update MemberInfo.</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete MemberInfo.</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Product Info.</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update Product Info.</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Order Product</td>
<td>R R C C C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update Order</td>
<td>R R W W W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search Product Info.</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search Delivery Info.</td>
<td>R R R R R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search Member Info.</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... ... ... ... ... ... ...

Fig. 8. An example of identified components for EC.
The resulting matrix after applying the clustering algorithm is shown in Table 7, with the labels arranged in a top-left-to-bottom-right diagonal. The regional boxes in the matrix should not overlap each other unless there is a conflicting relationship.

By using the resulting matrix, we define each cluster as one component. In this example of EC, three components are identified from the matrix — Customer Management, Order Management, and Product Management as shown in Fig. 8. Each component in the figure shows all the classes included in the component.

### 4.6. Allocating business objects into components (DA6)

Business objects typically contain persistent data and they are invoked by system-level objects or controllers. Hence, we need to assign business objects to appropriate components.

Since we have considered the relationships between use cases and classes in a previous task, we can now assign classes having a strong coupling with use cases into components in which the related use case is contained. If a class is transient (except for interface or abstract classes), we replicate it and assign it to a referencing component; otherwise the class is assigned to specific components and has dependence with another component. In the case of a class referencing by another component, we represent dependency relationships between components and other classes. If all classes in a component are referenced by another component, we indicate dependency relationships between components as in Fig. 9.

![Diagram](image)

**Fig. 9.** Dependency between components.

### 4.7. Defining component requirement specification (DA7)

After identifying and preliminarily designing components, we write component requirement specifications as in Fig. 10. The component requirement specification consists of the component name, a brief description, workflow description, contained classes, cross-references, friendly components, and C&V information. The
<<Component Requirement Specification>>

1. Component Name: Order Management Component
2. Brief Description: This component manages ordering products, containing ‘create order’, ‘search order’, ‘cancel order’, ‘process payment’, and ‘process delivery’ functions.
3. Work Flow
   3.1 Work flow for ‘Order Product’ Use Case
      3.1.1. After a customer adds one or more products into shoppingCart, he requests order.
      3.1.2. The system displays order information, and requests customer to inputs {payment}, {delivery}, and password(A1).
      3.1.3. A customer inputs {payment}, {delivery}, and password(E1).
      3.1.4. After verifying customer’s input information, the system computes delivery fee, and total amount.
      3.1.5. The system processes payment for computed total amount.
      3.1.6. After payment processing is confirmed, the system adds customer’s order information, decreases amounts of products, and displays payment information.
3.2 Work flow for ‘Update Order’ Use Case
4. Contained Classes: Order, Payment, Delivery, ShoppingCart
7. Commonality: Refer to Table 4.
8. Variability: Refer to Table 5.

Fig. 10. Component Requirement Specification for ‘Order Management’.

workflow describes the business logics at the conceptual level, and it captures event flows embedded in use cases. If a component contains several use cases, a workflow description is given for each use case.

4.8. Refining domain analysis model (DA8)

The purpose of this activity is to review and refine artifacts of the domain analysis phase by checking consistency among artifacts or completeness of each artifact. We propose guidelines for refining artifacts as follow:

— The set of requirement specifications collected in activity DA1 should be as complete as possible to be able to develop highly reusable components.
— In determining commonality, other factors such as influence of specific family members, sponsorship of developing components, and unknown potential family members should be considered in addition to the commonness of functions among members.
— Since components are evolved and used over a long period of time, the scope of variability should be determined in a way that newer variants can be effectively added in the future.
— In activity DA5 for identifying components, domain knowledge and experience
can also be used to define candidate components.

If a class is required by two or more components, the ‘read’ and ‘write’ usage
and frequency of message invocations on the class should be considered to decide
which component includes the class. In some cases, a separate component having
the class may be newly defined.

5. Preliminary Design (PD) Phase

The Preliminary Design phase is to add design-level details to the component re-
quirement specifications produced in the previous phase. This phase consists of six
activities which can be applied iteratively.

5.1. Identifying message flows (PD1)

A component may contain one or more use cases. Message flows between objects
or components in the domain analysis phase are not initially defined, so this task
extracts these flows between objects or between object and component for each
use case. To express message flows, we extend the UML sequence diagram with
component related elements as in Fig. 11. The message flow of sequence diagram

![Extended Sequence Diagram](image_url)

Fig. 11. An example of extended sequence diagram.
It is suggested to use a ‘controller’ object in this sequence diagram to show the coordination of messages between clients and system and to mediate the message flow among objects. The message flow, i.e. workflow, can be obtained from corresponding use case descriptions.

5.2. Defining class interface (PD2)

This activity is to add design-level details to the conceptual object model constructed during the previous phase. That is, we define all elements of class diagrams including class methods, navigability, multiplicity, constraint, and qualified association. Additional classes can be added if they are needed for implementation.

To define methods for classes, we refer to the object sequence diagram of the previous task. If one object receives a message from other objects, we map the message into the methods of the object, as shown in Fig. 12. Because one component consists of one or more objects, all message flows between objects should be included. These message flows are then reflected on the component interface.

5.3. Defining customization policy (PD3)

This activity is to define customization mechanisms for variability so that components can be tailored by component consumers. We need to consider customization
mechanisms for variant attributes, variant logics, and variant workflows. The cus-
tomization mechanism for variant attributes is further classified into two types, one
for variant values of attributes and the other for variant attributes having fixed
operations. Using the C&V table and detailed class diagrams, we define customiza-
tion methods of variant attributes as \textit{set(AttributeName)}. For example, if ‘color'
attribute of ‘Product’ class is identified during domain analysis, the customiza-
tion method will be \textit{setColor (Color color)}. Those set methods are assigned to a
corresponding class.

To determine customization policy for variant attributes having fixed operations,
we define those variant attributes as a template class. In this case, any object type
can be parameterized. For example, if there are intStack and Stack, we define the
stack class as a template class, as shown in Fig. 13. However, if there are different
attributes, we define those different attributes as a subclass.

\begin{figure}[h]
\centering
\includegraphics[width=0.3\textwidth]{fig13.png}
\caption{Defining variant attribute set as template class.}
\end{figure}

The customization policy for variant logic is classified into three types: single
discriminant, multiple discriminant, and option discriminant. A discriminant is
any feature that differentiates one system from another [15]. In COMO, we further
divide single discriminants and multiple discriminants into two types: abstract class
and interface.

Single discriminants are a set of mutually exclusive features, of which only one
can be used in a system, for example, all mobile phones (but displays can vary, such
as by the number of displayable characters). The single discriminants can then be
modeled as an inheritance hierarchy.

If the logics of some methods are different in applications, we define the base
class as an abstract class in which common features are modeled and specific fea-
tures are modeled as subclasses. Only one subclass can be instantiated in any single
system, and the set of subclasses is known as a realm. If required, the realm can
be extended for a new system. Figure 14 shows a single adapter pattern in which
the dotted box denotes a realm and the three dots within it indicate that the realm
can be extended in the future.

Sometimes, methods in a subclass must be accessible to other classes in the
model. In such cases, we place virtual functions in the base class (see operation
B in Fig. 14) so that other model parts can refer to base-class instances without knowing which subclass a given system will use. However, if the logics of all methods in subclasses are different, we define the base class as an interface, and the subclasses implement that base class. Multiple discriminants are a set of optional features that are not mutually exclusive; at least one must be used. For example, there must be at least one way of placing a call on a mobile phone, but there may be several ways, such as pressing the digits, pressing redial, or voice dialing.

We model the multiple discriminants in the same way as single discriminants: as an inheritance hierarchy, with generic features modeled in a base class and specific features modeled as subclasses. In this case, if the logics of some methods are different in their applications, we define the base class as an abstract class. Otherwise, we define the base class as an interface.

However, in this pattern, more than one subclass can be instantiated in any single system. Figure 15 shows a multiple adapter pattern in which each subclass
has one instance and the set of instances is held in a collection. To access methods in a particular subclass, other model parts can refer to a base-class instance without having to know which subclass a given system will use at a given time. We achieve this by identifying the subclass instances according to a name string or some other unique identifier and storing them in a collection.

Option discriminants are single optional features that may or may not be used. For example, mobile phones can have an Internet connection facility, but they do not require one.

When a variant logic is optional, we model it by creating two associated peer classes. We do not model optionality as inheritance, because inheritance cannot be optional. The associated classes must have a 0-1 relationship on at least one end. Figure 16 shows an example: class B is an optional class of class A; class A does not assume the existence of class B. Therefore, it can be reused whether or not class B is reused.

The customization policy for variant workflows identify variability tasks of domain analysis as \( \text{set}(\text{class name}::\text{method name}1, \ldots, \text{class name}::\text{method name } n) \). We add additional classes having this workflow customization method in a detailed class diagram.

For example, we defined the customization method for the `Job' variant attribute as `setJob(Job job)'. We also defined the customization method for the `process-Payment()' variant behavior as `processPayment(Payment payment)'. Examples of variant attributes and variant behaviors are described in Fig. 17.

5.4. Defining component interface (PD4)

An interface is a set of named operations that can be invoked by clients. This activity is to identify and define component interface and component methods that exist in component interfaces. Both provided interfaces and required interfaces should be defined in CBD. To define a provided interface, we examine the component interface for each class contained in a component. As shown in Fig. 18, if there are message flows sent from the controller to the classes, we define a component interface for each class and assign message flows into the component interface.

To determine a required interface, we select sending message flows from the controller class to an object of other components or select sending message flows from the object in a component to an object of another component. That is, if
there are message flows between the object and other components or message flows between the controller and other components in a sequence diagram, we define such interface of other components as a required interface of the target component as depicted in Fig. 19.
An example of defining the component interface is given in Fig. 20. The dotted rectangles represent methods of a required interface in a component, and the rectangular boxes represent methods of a provided interface in a component.

After defining the component interfaces, we constructed a component diagram as shown in Fig. 21.
5.5. Defining component specification (PD5)

The component contract provides what component users need to use components and what a component developer needs to implement interfaces of components. In order to define the component contract, we must also define pre-condition, post-condition, and exceptions for each method of the component interfaces. In the component contract, customization methods are provided. Therefore, component users easily customize component’s attribute, logic, or workflows through the component’s customization methods.

In this paper, we identify the component contract as an extended IDL of CORBA component model [2] because IDL of CORBA is independent of specific programming languages (see Fig. 22).

After defining the component contracts, we can construct component specification. As illustrated in Fig. 23, a component specification consists of the component name, a brief description, the participants, a static model, a dynamic model, and the component contract sections.

In the participants’ section, the classes' name contained in a component is described. The component diagram is represented in the static model section, the sequence diagrams are described in a dynamic model section, and the component contract is illustrated in the component contract section.

5.6. Refining preliminary design model (PD6)

The purpose of this section is to review the artifacts of the preliminary design phase. First, we check for consistency among artifacts and synchronize them according to
<Component Contract>
Component: 'Order Management'

Provides:

Interface “IOrder” {
    exception nullShoppingCart;
    public Order();
        pre-condition: userID, password is valid.
        post-condition: order object should be created.
        raises (nullShoppingCart)
    public int computeTotalFee();
        pre-condition: payment and delivery is valid.
        post-condition: totalFee should be returned.
    public void processOrder();
        pre-condition: processPayment should be returned.
        post-condition: customer’s orderlist should be increased.
                        product’s amounts should be decreased.
                        shoppingCart should be deleted.
}

Interface “IPayment” {
    exception nullPayment;
    public Payment();
        pre-condition: payment information should be inserted.
        post-condition: payment object should be created.
        raises (nullPayment)
    public boolean checkPayment();
        pre-condition: payment should be created.
        post-condition: return value should be true.

    ...
}

Fig. 22. Component contract.

<Component Specification>
Component Name:
Brief Description:
Participants: // Contained Classes
Static Model: // Component Diagram
Dynamic Model: // Sequence Diagram
Component Contract: // Component Contract

Fig. 23. Component specification.

the following rules:

— The number of identified interfaces is mapped into the component interface of
the component diagram.

— Portions of the component requirement specification in the domain analysis
6. Case Study

In this chapter, we present a case study of applying COMO in developing components for electronic commerce (EC) domain. This domain has a high degree of commonality in the areas of customer management, order processing, payment processing, product management, and delivery processing.

6.1. Domain analysis phase

In the domain analysis phase, we identified the domain requirement set by gathering application requirement specifications, extracting common requirements in the electronic commerce domain from the domain requirement set, identifying components, and defining the component requirement specification.

6.1.1. Identifying domain requirement set

In order to identify the domain requirement set, we gathered application requirement specifications. We also defined imaginary application requirement specifications by interviewing domain experts and simulating existing application systems. The identified domain requirement is illustrated in Fig. 24.

We identified the term dictionary and function dictionary based on the domain requirement set. Our term dictionary is depicted in Table 2 and function dictionary is depicted in Table 3. In the domain requirement set of the electronic commerce domain, we used some duplicated terminologies, so we defined these duplicated terminologies as general and descriptive terminology in the term dictionary.

After setting our term dictionary, we created a function dictionary for electronic commerce domain as well. We did this because the function names are different in application requirement specifications, and it is therefore difficult for us to use common functions from the domain requirement set. Also, the function’s sizes are represented differently according to the application requirement specification. Therefore, we defined general function names used in the electronic commerce domain as domain function names. After establishing our function dictionary for the electronic commerce domain, we refined the functional requirements of each application requirement specification based on this dictionary.

6.1.2. Extracting commonality

In this section, we extracted common functions from normalized functions described
Requirement Specification for Lotte Shopping Mall

Functional Requirements

1. Joining Membership
   If a customer requests “Join membership”, the system displays “Membership format” menu.
   A customer inputs name, age, sex, nationality, rrr, email address, home information(zip code,address, phone number),
   office information(zip code, address, phone number), delivery option, credit card option.
   If the entered information is correct, the system displays “Welcome to our Membership” message.
   Otherwise, the system displays “Incorrect information” message.

2. Searching/Updating Member
   A customer inputs user id and password in order to search or update member information.
   The system searches user’s id and password, and requests changing option(password update,
   address/phone/email update, credit card update).
   If input information is corrected, the system displays corresponding information.
   A customer updates information, the system displays “information is updated”.

3. Searching Products
   If a customer enters product information to search, the system searches products matching input keyword,
   and displays searched product information(product name, product price).

4. Viewing Detailed Product Information
   If a customer clicks product, the system displays detailed product information(product name, price, amounts, description)
   and delivery information(delivery duration, location and fee, sales of Lotte Card).
   A customer can zoom in product image or add product items into shoppingBack.

Fig. 24. Domain requirement set of EC.

• Grouping functions into Use Cases
  = U1. Join Membership
  = U2. Search Member Info.
  = U3. Delete Member Info.
  = U4. Update Member Info.
  = U7. View Shopping Cart
  = U8. Update Shopping Cart
  = U9. Delete Shopping Cart
  = U10. Add Item into Shopping Cart
  = U11. Order Product

  = U12. Search Order
  = U13. Update Order
  = U14. Cancel Order
  = U15. Deliver Order
  = U16. Update Deliver
  = U17. Cancel Deliver
  = U18. Search Delivery Info.
  = U19. Return Product
  = U22. Update Product Info.

Fig. 25. Identified use cases.

in the domain requirement set. We identified common functions as shared functions
among the application requirement specifications, and the common functions we
identified are depicted in Fig. 4.

We assigned the degree of commonality in shared common functions according
to their intersected application amounts. For example, if a function is shared by all
application requirements, we assign it into high class as in Fig. 4.
We then identified four actors from the common requirements and use cases in the electronic commerce domain: Customer, Administrator, Banking System, and Supplier. After identifying these actors, we identified use cases by grouping cohesive functions as in Fig. 25. Then, we drew a use case diagram as in Fig. 26, based on identified actors and use cases.

We also constructed use case descriptions for each use case; one of the 'Order Product' use case is depicted in Fig. 5.

6.1.3. Identifying variability

Prior to identifying variability, we transformed the use case descriptions into structured use case descriptions as in Fig. 27. As described in domain analysis, the purpose of this step is to assist the component analyst in understanding and identifying variant parts of the component.

We then identify variability based on these structured use case descriptions. In order to identify variability, we identify attributes, logics, and workflows for each application; identify common attributes, logics, and workflows; and identify variant attributes, logics, and workflows as in Table 4.
Structured Use Case Description

Main Flow
1. getShoppingCart()
2. checkPassword() – E1
3. checkPayment() – E2, A1
4. checkDelivery() – E3, A2
4. computeDeliveryFee() – A3
5. computeTotalFee()
6. processPayment() – A4
7. addOrderList()
8. decreaseAmount()
9. deleteShoppingCart()

Alternative Flows
A1.1: checkPayment(CreditCard creditCard)
A1.2: checkPayment(OnLine online)
A2.1: checkDelivery(Customer customer)
A2.2: checkDelivery(String other)
A3.1: computeDeliveryFee(Ship)
A3.2: computeDeliveryFee(Airline)
A3.3: computeDeliveryFee(Land)
A4.1: processPayment(CreditCard)
A4.2: processPayment(Online)

Exception Flows
E1: throw InvalidPasswordException()
E2: throw InvalidPaymentException()
E3: throw InvalidDeliveryException()

Fig. 27. Structured use case description.

Examples of variant attributes include payment type and delivery type in the ‘Order Management’ component and variant logics include ‘processPayment()’ or ‘computeDeliveryFee()’ in the ‘Order Management’ component.

We also determined the type, scope, default value, and determination value of variability and assigned values of each variability into the variability list as in Table 5.

6.1.4. Constructing conceptual object model

In this section, we constructed a conceptual class diagram by identifying business objects including ‘Order’, ‘Customer’, ‘Product’, ‘Delivery’, ‘Payment’, ‘ShoppingCart’, their attributes, and their relationships with each other. In this domain, we identified five classes (e.g. ‘Order’, ‘Customer’, ‘Product’, ‘Delivery’, ‘Payment’, ‘ShoppingCart’).

6.1.5. Identifying components

In this section, we identified components by applying the proposed clustering algorithm to identified use cases and classes. First, we made use of the case/class
matrix based on use cases and classes and assigned the relationship value in each cell of the matrix.

We applied the clustering algorithm into Table 6, and the clustered result is shown in Table 7, in which use cases and classes having strong coupling are clustered. We define each cluster as one component, and we discovered: ‘Customer Management’, ‘Order Management’, and ‘Product Management’.

Identified components are shown in Fig. 8. Each component contains one or more classes.

6.1.6. Allocating business objects into components
In this task, we allocated business objects modeled in Fig. 8 into identified components. A dependency relationship is described among the ‘Order Management’, ‘Customer Management’, and ‘Product Management’ components because the ‘Order Management’ component refers to the ‘customer’ class of ‘Customer Management’ and the ‘product’ class of ‘Product Management.’ The result of allocated business objects in these components is depicted in Fig. 8.

6.1.7. Defining component requirement specification
In this task, we defined component requirement specifications for each component. For example, we described component requirement specifications of ‘Order Management’ in Fig. 10.

6.2. Preliminary design phase
6.2.1. Identifying message flows
In this task, we identified message flows for each use case based on each use case description, and we represented the message flows in a sequence diagram as in Fig. 11.

6.2.2. Defining class interface
In this section, we defined methods of classes based on identified message flows. As we mapped message flows between objects into a method of classes, we constructed a detailed class diagram. That is, we elaborated on the conceptual class diagram of the domain analysis phase by adding methods of classes as in Fig. 12.

6.2.3. Defining customization policy
In this section, we determined customization policies for identified variant parts (e.g., variant attributes, variant behaviors, variant workflows). In this domain, we identified variant attributes and variant behaviors in the domain analysis phase,
and we defined customization methods for variant attributes and variant methods. For example, we defined the customization method for ‘Job’ variant attribute as ‘setJob(Job job)’, and the method for ‘processPayment( )’ variant behavior as ‘processPayment(Payment payment)’. Examples of variant attributes and variant behaviors are described in Fig. 17.

6.2.4. Defining component interface

In this task, we identified a component interface for each component, and represented each component interface in a sequence diagram. An example of defining a component interface is illustrated in Fig. 20. In the diagram, a dotted rectangle indicates a required interface and rectangular boxes indicate provided interfaces.

In this case study, we identified six component interfaces, and we defined public methods in each identified component interface. After defining the component interfaces, we constructed component diagrams as shown in Fig. 21.

6.2.5. Defining component specification

In this task, we defined component specifications and component contracts for each component. An example of the component contract for the ‘Order Management’ component is illustrated in Fig. 22.

7. Assessment

With various experiments and the component reference model, we assess the proposed component methodology with existing CBD methodologies. We compare existing methodologies to the proposed methodology through the comparison criteria proposed in [16].

According to Object Agency [16]’s approach, we evaluate our methodology based on a framework consisting of a series of questions used to identify and quantify the methodology’s support for a specific development process. This framework considered three major areas of each methodology: concepts, notations, and process. In this paper, we classify process factor into artifact’s specification level and details of activity description.

7.1. Comparison on meta-model

We compare existing methodologies to our proposed methodology according to the comparison criteria. Table 8 shows the result of the comparison of methodologies based on the structural meta-model of each component reference model.

The comparison criteria in column 1 include a set of elements vital to constituting a component reference model. As shown in Table 8, CBD96, Catalysis, Fusion, UNIFACE, and SCIPIO support most of the elements, but only the proposed methodology, COMO, supports all the key elements including customization methods and component workflows.
Table 8. Comparison based on concept.

<table>
<thead>
<tr>
<th>Elements</th>
<th>Methodology</th>
<th>CBD96</th>
<th>Catalysis</th>
<th>Fusion</th>
<th>UNIFACE</th>
<th>SCIPIO</th>
<th>COMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Interface</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Class</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Method</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Customization Method</td>
<td>Partial</td>
<td>Partial</td>
<td>Partial</td>
<td>Partial</td>
<td>Partial</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Component Workflow</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Component Specification</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Class Relationship</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Component Relationship</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The comparison criteria in column 1 include a set of elements vital to constituting a component reference model. As shown in Table 8, CBD96, Catalysis, Fusion, UNIFACE, and SCIPIO support most of the elements, but only the proposed methodology, COMO, supports all the key elements including customization methods and component workflows.

7.2. Comparison on Notations & Expressiveness

We also compare methodologies based on notation criteria. The result of that comparison, as described in Table 9, indicates that almost all methodologies support UML’s syntax and semantics. However, catalysis and our proposed methodologies extend UML’s syntax.

As shown in Table 9, Catalysis methodology uses not only all of UML’s diagrams, but also extensional diagrams. Because the Fusion and UNIFACE methodologies use component diagrams of UML, component developers have found it hard to represent component’s semantics sufficiently. As shown in Table 9, COMO produces an extend set of notations.

Table 9. Comparison on notations.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Methodology</th>
<th>CBD96</th>
<th>Catalysis</th>
<th>Fusion</th>
<th>UNIFACE</th>
<th>SCIPIO</th>
<th>COMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax Defined</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Semantics Defined</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Component Diagram</td>
<td>Extended</td>
<td>Extended</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Extended</td>
<td></td>
</tr>
<tr>
<td>Class Diagram</td>
<td>Extended</td>
<td>Extended</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Use Case Diagram</td>
<td>Yes</td>
<td>Extended</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Interaction Diagram</td>
<td>No</td>
<td>Extended</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
### Table 10. Comparison on artifacts and specifications.

<table>
<thead>
<tr>
<th>Methodology</th>
<th>CBD96</th>
<th>Catalysis</th>
<th>Fusion</th>
<th>UNIFACE</th>
<th>SCPIPO</th>
<th>COMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component Req. Spec.</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Use Case Model</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Behavior Specification</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Class Diagram</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Component Diagram</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Interaction Diagram</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Component Contract</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Component Specification</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Totals</td>
<td>26</td>
<td>27</td>
<td>18</td>
<td>19</td>
<td>22</td>
<td>28</td>
</tr>
</tbody>
</table>

### 7.3. Comparison on artifacts and their specifications

We then evaluate methodologies based on deliverables generated from the development process. The deliverables for each method are evaluated using the following criteria:

- Level 0 indicates that the artifact is not used.
- Level 1 indicates that the artifact is used, but its definition is not given.
- Level 2 indicates that the artifact is used, and its definition is given, but instructions are not given.
- Level 3 indicates that mention is made, a definition is supplied, and instructions are defined.

Existing methodologies do not provide concrete modeling guidelines (see Table 10). However, our proposed methodology provides detailed guidelines as well as deliverables.

As shown in Table 10, COMO provides the most complete definitions and instructions on artifacts.

Next, we compared methodologies based on the details of activity description. In some cases, these heuristics are simple and obvious, while others are less obvious. The availability of a large set of heuristics simplifies the process execution.

Table 11 represents a subject assessment of the number of heuristics presented by the steps of identifying commonality, variability, components, classes, operations, and interfaces. These steps are not exhaustive, but they serve as an indicator to the degree of support the method provides for heuristics. A “1” indicates few, if any, heuristics, while a “3” indicates three or more heuristics.
Table 11. Comparison on the details of activity description.

<table>
<thead>
<tr>
<th>Methodology</th>
<th>CBD96</th>
<th>Catalysis</th>
<th>Fusion</th>
<th>UNIFACE</th>
<th>SCIPIO</th>
<th>COMO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifying Commonality</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Identifying Variability</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Identifying Components</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Identifying Provided Interface</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Identifying Required Interface</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Designing Attribute Customization Method</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Designing Logic Customization Method</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Designing Workflow Customization Method</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Describing Component Contract</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Describing Component Specification</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Designing Platform-based Component</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Totals</td>
<td>15</td>
<td>19</td>
<td>14</td>
<td>15</td>
<td>15</td>
<td>21</td>
</tr>
</tbody>
</table>

7.4. Traceability among artifacts

There are many artifacts produced in the domain analysis phase and preliminary design phase during the component modeling process. All of the activities require inputs and outputs to process their work, and a set of related artifacts should be traceable in a well-defined methodology. Therefore, we depict traceability between inputs and outputs for each activity in each phase. Figure 28 shows the relationships among artifacts and depicts how elements of each artifact are used in other artifacts. For example, the list of components in Fig. 28 is traced to use case diagram in terms of use case and to class diagram in term of class.

8. Concluding Remarks

We proposed a practical domain analysis and component modeling methodology, COMO, that extends current CBD methodologies and adopts UML foundation. Each phase consists of activities, and each activity is given instructions, artifact templates, and examples. In this paper, we focus on two front-end phases: domain analysis and preliminary design. Domain analysis phase consists of eight activities; identifying domain requirement set, extracting commonality, identifying variability, constructing conceptual object model, identifying components, allocating business objects into components, defining component requirement specification, and refining domain model.

Preliminary design phase consists of six activities; identifying message flows, defining class interface, defining customization policy, defining component inter-
face, defining component specification, and refining preliminary design. We paid particular attention to modeling issues related to component development because they indicate how to identify components, how to identify commonality and variability of components, and how to identify components’ interfaces. The main result of this paper is a new approach to presenting modeling information on how to develop components, comprising a set of usage modeling guidelines.

In future work, we plan to develop guidelines for component implementation and testing, integrate component development process into component-based software development, and incorporate proposed techniques into component modeling tools.

References


