

## Component Identification Methods Applying Method Call Types between Classes

MISOOK CHOI AND EUNSOOK CHO\*

*Department of Computer Engineering*

*Woosuk University*

*Chonbuk 565-701, South Korea*

*E-mail: khc67\_kr@hanmail.net*

*\*Department of Software*

*Seoil College*

*Seoul 131-720, Korea*

*E-mail: escho@seoil.ac.kr*

Identifying reusable and autonomous components is one of the most important and difficult tasks in developing component-based systems. However, the existing component development methodologies do not provide a clear standard for component identification and depend on the intuition and experience of individual developers. As a result, people with limited expertise cannot easily identify components. This paper proposes methods to identify components based on object-oriented techniques including Use Case diagrams, Class diagrams, and Sequence diagrams. We use two distinct steps in identifying components, that is, system and business component identification. In particular, our proposal considers dynamic dependency characteristics of the method call types and directions. Case study and assessment with the existing identification approaches help to verify the practicality of our proposal.

**Keywords:** component identification, method call types between classes, system component, business component, dependency between classes

### 1. INTRODUCTION

Due to the rapid changes and complexities involved in information technologies, developing non-trivial software from scratch is no longer an attractive option unless it is absolutely necessary. Since early 1990s, component-based software development emerged as a viable alternative to the conventional software development paradigms by being highly cost-effective and offering significantly reduced time to market. This methodology hinges upon the idea of assembling independent software parts (namely, components) featuring a set of well-defined functionalities into a single system to satisfy the constantly changing user requirements more effectively and to still maintain the quality of the software product. A component is a unit of software that can be developed and distributed in an independent manner. In addition, it is a highly cohesive software package that can be easily combined with another component without requiring fundamental changes in its source code [1]. Therefore, one of the most important factors in component based software development is the ability to identify an independent component that is

---

Received July 1, 2005; accepted November 24, 2005.  
Communicated by Sung Shin.

reusable and exhibiting minimal coupling for improved maintenance. The existing methodologies [2-5] for identifying components rely heavily on the intuitions and experiences of an individual developer. They lack more precise criteria and methods. Furthermore, a majority of them address only business component identification although a component-based software architecture consists of system components (building blocks of a subsystem) and business components that use the functionalities of the system components [2-8]. Therefore, this paper proposes domain model-based criteria and methods to easily identify components. More specifically, our proposal extends the existing solutions and overcomes their deficiencies using Use Case, Class, and Sequence diagrams. One of the salient features of our component identification approach is the identification of system components (constituting a subsystem) prior to business component. One can more effectively identify system components since there are less number of classes in the subsystem compared with the number of classes found in the entire system. Furthermore, our identification approach takes advantage of the structural characteristics of class relationships and the subservient nature of static and dynamic relationships stemming from the method call patterns and directions as well as assigning certain weight quantifying degree of dependency to each relationship. To prove that our approach is more effective in identifying components, we conduct an analysis and compare its results with those of others through our case study exploring an Internet shopping mall problem in the domain of electronic commerce.

## 2. RELATED RESEARCH

### 2.1 System and Business Components

There are two types of components in a component-based system. One of them is a system component that encapsulates a set of similar, reusable functionalities. The other is a business component that utilizes the functionalities of the system components [4]. A *system component* is a unit of functional reuse. It is also a unit of a subsystem. Since system components are identified around a Use Case embodying a collection of similar functionalities, one can think of each Use Case as an interface through which a system component fulfills its functionality. *Business components* are identified by grouping closely related classes designed to deliver the system component functionalities of Use Cases. That is, a business component is an independent software part one can use to execute the functionality defined by the interfaces of system components.

### 2.2 Existing Identification Methods for Components

The CBD96 [3] method of the Cool Joe product by Computer Associates, Inc. adopts a business component identification method that groups closely related objects based on core types. Their method does not factor in the concept of reusable system components and suffers from the same problems pointed out earlier (dependency on the intuitions of individual developers and mandatory consideration of the entire domain).

The Advisor method (by Cheesman and Daniels) [4] is an extension of the CBD96 method. It clearly defines what a system component is but treats system and business

components in an isolated fashion. This results in a less-than-optimal effort to analyze Use Cases for determining the class membership to a system component because one still needs to separately decide the class membership to a business component. Analogously with other existing approaches, business component identification in this approach aggregates closely related objects according to core types and is dependent on the intuitions and experiences of an individual developer.

IBM's Rational Unified Process (RUP) [5] integrates three different methodologies originally developed by Booch, Rumbaugh, and Jacobson. Component identification in RUP is primitive and does not offer any concrete guidelines or processes. As a result, RUP is as ad-hoc as any other existing approaches in identifying components.

ETRI's Marmi III [6] uses object models and Use Case Data Access (UDA) tables for its component identification. The object model-based identification finds its core classes among entity classes and extracts candidate components by grouping them along with other related boundary classes. This method is similar with the CBD96 method mentioned above. UDA table-based identification assigns different weights to the Create, Read, Write, and Delete methods of a class associated with a Use Case and analyzes the degree of coupling. However, Marmi III does not consider method call patterns and directions. Rather, it only analyzes relationships between Use Cases and Classes. Consequently, Marmi III's component identification method is not as precise as it can be. It also assumes that Use Cases only referencing classes can become part of any candidate component because the condition for a stronger association between a class and a Use Case is not satisfied if a Use Case needs to just reference a class for accomplishing a desired functionality. Identifying system components prior to business components can eliminate this ambiguity, but Marmi III does not have the concept of system components and identifies its components from the entire domain.

Lee *et al*'s method [7] does not consider the method call patterns and directions and therefore fails to precisely identifying component.

### 3. COMPONENT IDENTIFICATION STANDARDS

We classified the characteristics of dependency relationships between classes for an object-oriented domain model in terms of their static and dynamic aspects. This section proposes standards to identify components and the proposed dependency characteristics between classes are then applied to our component identification standards.

#### 3.1 Classification of the Characteristics of Static and Dynamic Relationships between Classes

A dependency relationship (formally represented as  $\langle C2, C1 \rangle \in R$ ) exists between two classes when  $C2$  sends a message to  $C1$ . In other words,  $C1$  depends on  $C2$  when  $C2$  influences  $C1$  by making a method call to  $C1$ . This subsection classifies these dependency relationships in terms of static structural relationships, dynamic message call types, and message call directions.

### 1) Static dependency resulting from structural characteristics between classes

Possible structural relationships between classes are composition (aggregation), inheritance, and association.

In a composition (aggregation) relationship (that is, class *A* contains classes *B* and *C*), writing to class *A* directly affects classes *B* and *C*. Invoking an operation on an instance of class *A* also has a direct impact on the instances of classes *B* and *C*. For example, if an instance of class *A* is created, deleted, written, or read, the same operation is applied to the instances of classes *B* and *C*. Also, the classes in a composition (aggregation) relationship tend to be operated as a functional unit. In particular, if the two classes belong to different components, respectively, the composition relationship is valid with the two components because the two classes in the composition relationship are within same life cycle.

In case of the inheritance relationships between classes, the classes are highly cohesive because there is a reusable relationship with child class that inherited member variable and method from parent class. Thus, if the classes in inheritance relationship are distributed into several components, dependency between the components should be higher.

Therefore, for classes participating in composition or inheritance relationships, changes in the instances of classes (one or more level) higher in a class hierarchy have a direct and strong effect on instances of lower level classes, resulting in a strong bond that can be regarded as a master-slave relationship. This type of classes cannot be part of different components and must be the members of the same component because components are meant to be independent from each other and shielded from each other's influences with minimal coupling. However, if class hierarchy layers in relation to the composition (aggregation) or inheritance relationships between classes are complicated due to the designing problems of the system in itself, a designer can properly distribute the classes into each component.

For classes in association relationships (that is, if there exists an association between classes *A* and *B*), there exists a peer relationship through message passing. In other words, if  $\langle A, B \rangle \in R$  and *R* is an association relationship, then an instance of class *A* can send an instance of class *B* a message to create, delete, write, and read, or an instance of class *A* can invoke a method on an instance of class *B*. In an association relationship, a class does not have a direct influence on another class as in composition (aggregation) and inheritance relationships but has a varying degree of dependency. Therefore, depending on the type of a method call, it is possible to have a completely independent relationship between classes. Consequently, the strength of dependency between classes based on their structural characteristics is in the order of composition (aggregation) > inheritance > association.

### 2) Dynamic dependency resulting from message call types between classes

Dependency relationships between classes can also be established through method call types. To evaluate the degree of dependency between classes, this paper classifies method call types as follows.

1. Object  $A_a$  of *A* class calls a method that creates object  $B_b$  of *B* class;

2. Object  $A_a$  of  $A$  class calls a method that deletes object  $B_b$  of  $B$  class;
3. Object  $A_a$  of  $A$  class calls a method that writes to object  $B_b$  of  $B$  class;
4. Object  $A_a$  of  $A$  class calls a method that reads object  $B_b$  of  $B$  class;
5. Object  $A_a$  of  $A$  class calls a method that contains one of the methods in 1, 2, 3, and 4.

In the method call types 1 and 2, other objects using object  $B_b$  are structurally affected since object  $A_a$  changes the structure of object  $B_b$ . In the type 3, other objects using object  $B_b$  are affected in terms of the values of their attributes since the attributes of an object are modified unlike in the types 1 and 2 that involve structural changes. In the type 4, other objects using object  $B_b$  are not affected at all since there is no structure (as in the types 1 and 2) and attribute changes (as in the type 3). To be more specific, when object  $A_a$  in component  $C_a$  sends a message to read object  $B_b$  in component  $C_b$ , the dependency relationship of the two components is weaker than that of the components in the types, 1, 2, and 3. In the type 5, the degree of dependency varies depending on what method out of 1, 2, 3, and 4 is invoked by another method.

### 3) Dynamic dependency resulting from message directions

If  $\langle A, B \rangle \in R$  and object  $A$  sends a message to create object  $B$ , then  $B$  is dependent on  $A$ . The creation of  $B$  structurally affects objects manipulating  $B$ . However, if object  $A$  sends a message to read object  $B$ ,  $A$  does not affect  $B$  at all because  $A$  is simply using the data of  $B$ . The direction of message calls changes a dependency relationship. Therefore, one must consider the direction of message calls.

1.  $A \rightarrow B$ :  $A$  invokes a method on  $B$ ;
2.  $A \leftrightarrow B$ :  $A$  and  $B$  invoke methods on each other.

In 1, the degree of dependency relies on what type of a method  $A$  invokes on  $B$ . In 2, although the existing research [7] defines that there is unconditionally strong dependency in the presence of bi-directional method calls, this paper assumes that the degree of dependency still varies depending on the method call types.

## 3.2 Standards for Identifying Business Components

According to the dependency characteristics presented in the subsection 3.1, we define the criteria for identifying business components as follows:

**Criteria 1** Classes participating in composition or aggregation relationship must belong to the same component since there is a strong dependency relationship. However, if the class hierarchy layer between the classes in the components created is complicated because depth between the classes in composition or aggregation relationship is excessive, a designer is able to properly distribute these classes into each different component.

**Criteria 2** Classes participating in inheritance relationship must belong to the same component since there exists a strong dependency relationship. Especially, abstract class among the classes in inheritance relationships or interface is widely reused by other classes; thus the classes in this type of relations do not need to be included in one com-

ponent. However, if the inheritance relationship between classes, except the abstract class or interface, is existed, it should be included in same component. On the other hand, if the depth of classes in inheritance relationship, except the abstract class or interface is more excessive, a designer should properly distribute the classes in class hierarchy layer with due degrade to the sharing degree between classes.

**Criteria 3** Classes participating in an association relationship whose method call types is either creation or destruction, must belong to the same component since there exists a strong dependency that changes the structure of objects.

**Criteria 4** Classes participating in an association relationship whose method call type is read, can be members of different components that maintain their independence from each other since there exists a weak dependency relationship.

**Criteria 5** Classes participating in an association relationship whose method call type is modification can belong to the same component or spread into multiple components.

**Criteria 6** Domain knowledge and experience can also be used with applying the criteria 1, 2, 3, 4, or 5 defined above.

### 3.3 Standards for Identifying System Components

Most of the existing methods for identifying components do not offer any concrete guidelines and expose a component by grouping (semantically) similar Use Cases. However, Bunge [11] decides the membership of a component based on similarity, which is defined by a set of common variables read by any given two methods. Furthermore, the functional similarity of Use Cases is decided by a set of common objects referenced by different Use Cases. Using a similar approach, this paper proposes identifying system components by considering actors and objects associated with a Use Case as well as Use Cases themselves.

Accordingly, to identify system components by grouping UseCases, we define the criteria for identifying system components as follows;

**Criteria 1** The usecases manipulating a common set of attributes can be grouped.

**Criteria 2** Closely related usecases by function can be grouped.

**Criteria 3** The usecases in the same functional category can be grouped.

**Criteria 4** The usecases of the include and extends relationships can be grouped.

**Criteria 5** The domain knowledge and other domain-specific criteria can be applied.

### 3.4 Standards to Reallocate Common Classes between System Components

We also reallocate classes used by more than one system component using the dependency relationship characteristics discussed in subsections 3.1 and 3.2. With this ap-

proach, one can identify a business component using the exposed system components since they don't belong to more than one system component. One can also easily understand the entire system and identify business components more effectively since business components are exposed using system components as central ingredients that are the subset of the whole system.

Table 1 defines how one can apply the dependency relationship characteristics between classes to reallocate common classes shared by multiple components.

**Table 1. Criteria to be applied to reallocate common classes.**

Criteria	Relationship between common class and system component 1	coupling	Relationship between common class and system component 2	Arrangement of Common Classes
Criteria 1	Composition	>	Association	Component 1
Criteria 2	Aggregation	>	Association	Component 1
Criteria 3	Inheritance	>	Association	Component 1
Criteria 4	Create	>	Delete, Write, Read	Component 1
Criteria 5	Composition, Aggregation, Inheritance	∪	Composition, Aggregation, Inheritance	Component 1 + Component 2

#### 4. THE OVERALL PROCESS AND METHODS TO IDENTIFY EFFICIENT COMPONENTS

This section proposes the overall process and method to identify business components based on system component. Therefore, the overall process to identify components consists of two phases as shown in Fig. 1; *System component identification phase*, *Business component identification phase*.

##### 4.1 Process and Method to Identify System Components

Below we propose the steps to analyze and identify system components.

- Step 1:** Analyze requirements and develop Use Cases, which leads to the completion of a requirements analysis model.
- Step 2:** Expose objects using Use Case realization process of RUP according to the scenarios of each Use Case (that is, the flow of events).
- Step 3:** Complete class and sequence diagrams using the objects exposed by the realization process.
- Step 4:** Group Use Cases and identify system components by considering the similar functionality of Use Cases and the degree of referencing the same classes from the perspective of a system service.
  - 4.1** Group Use Cases into a single Use Case when they participate in a composition (aggregation) relationship.

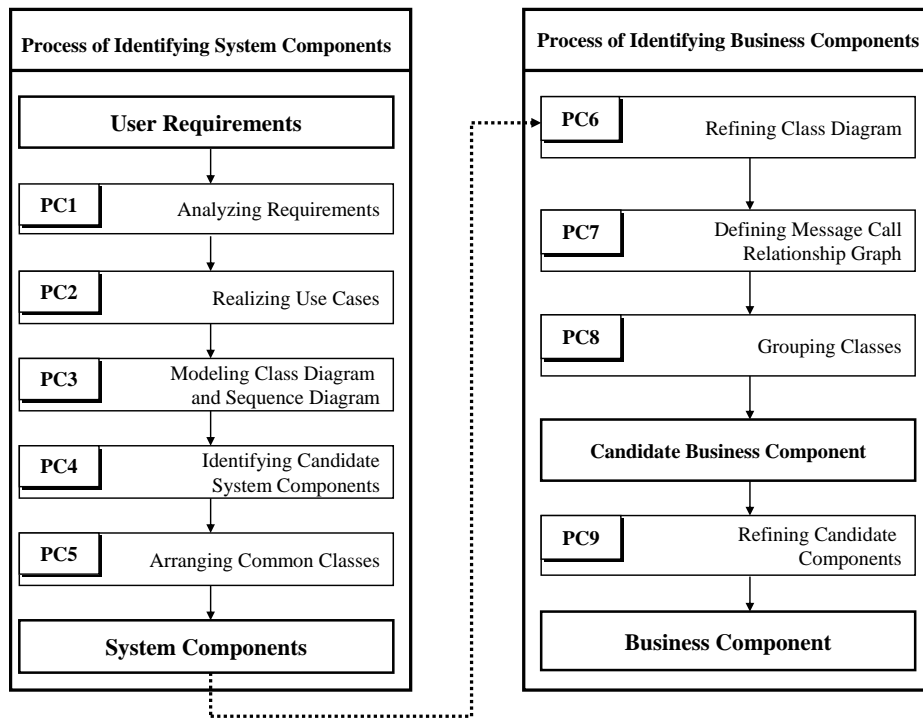


Fig. 1. Component identification process.

- 4.2 Create a table showing the relationship between Use Cases for each Actor and entity objects participating in them.
  - 4.3 Group Use Cases by considering similarity in functionality and the degree of common membership for objects associated with different Use Cases (utilizing the table in step 4.2) and define the newly grouped Use Cases as system components.
  - 4.4 Group Use Cases using the domain knowledge and other domain-specific criteria.
- Step 5:** Identify business components based on system components that do not share any common classes. Reallocate common classes using the rules in Table 1 if components share common classes.
- 5.1 Create a matrix table applying the CRWD (Create, Read, Write, and Delete) method call types between Use Cases and classes by considering a sequence diagram.
  - 5.2 Decide the reallocation of common classes according to the common class reallocation rules 1, 2, 3, and 4 if the common classes are not strongly coupled with each system component at the same time.
  - 5.3 Consolidate the system components and identify business components using the common class reallocation rule 5 if the common classes are strongly coupled with each system components at the same time.

**Table 2. Identified system components and common classes in them.**

Actor	UseCase	Classes										
		C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
Actor 1	Usecase 1	√		√								
	Usecase 2	√		√								
Actor 2	Usecase 3		√	√	√							
	Usecase 4		√	√	√							
	Usecase 5			√	√							
	Usecase 6				√	√	√			√	√	
Actor 3	Usecase 7				√	√	√					
	Usecase 8				√	√	√			√	√	√

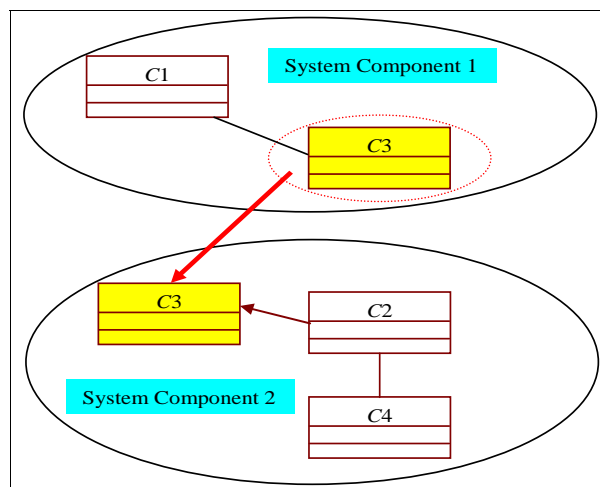


Fig. 2. Reallocating common classes present in identified system components.

#### 4.2 Process and Method to Identify Business Components

Next we show the methods and steps to identify business components consisting of system components. Since the system components identified using the steps introduced in subsection 4.1 do not share any common classes, it is now possible to identify business components using the exposed system components. We identify business components by grouping classes in terms of their dependency characteristics discussed in subsections 3.1 and 3.2.

**Step 6:** Refine class diagrams by grouping classes that have strong dependency relationships such as composition (aggregation) and inheritance as described in subsections 3.1 and 3.2.

**Step 7:** Define a message call relationship graph based on the corresponding sequence and class diagrams derived from step 6.

**Step 8:** Group classes participating in message call types (subsections 3.1 and 3.2) such as create and delete into one class. Each grouped classes are identified as one candidate business component.

**Step 9:** Further refine the candidate components and finalize the business component identification process considering the characteristics described in subsections 3.1 and 3.2.

## 5. CASE STUDY AND EVALUATION

The field of electronic commerce applications is rapidly growing due to the rapid development in the Internet and Web application techniques. This paper selects the electronic commerce domain to provide examples for component identification and validates the effectiveness of our approach.

### 5.1 Case Study

**Step 1:** Analyze requirements, develop Use Cases, and complete a requirements analysis model as shown in Fig. 3.

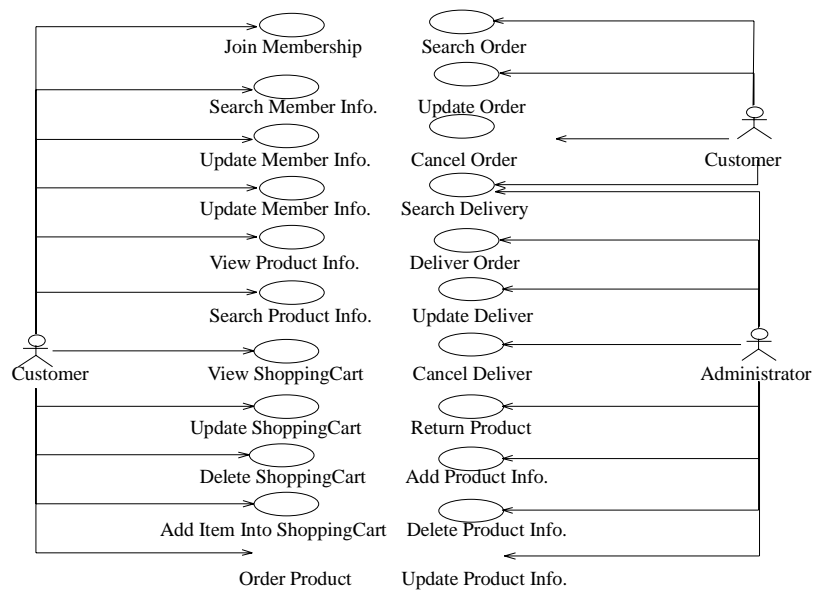


Fig. 3. Use Case diagram.

**Step 2~3:** Expose objects through a realization process of RUP according to each Use Case scenario and complete the corresponding sequence and class diagrams as shown in Figs. 4 and 5.

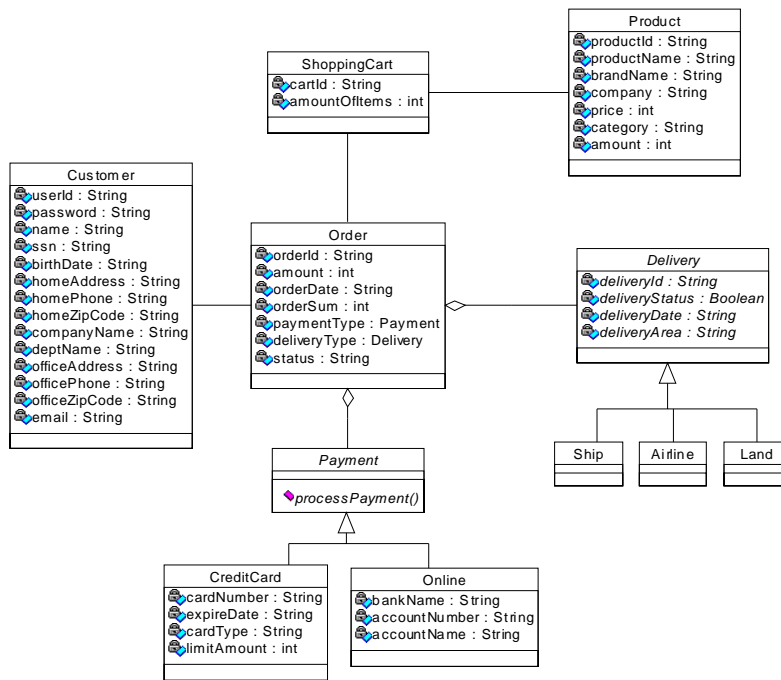


Fig. 4. Class diagram.

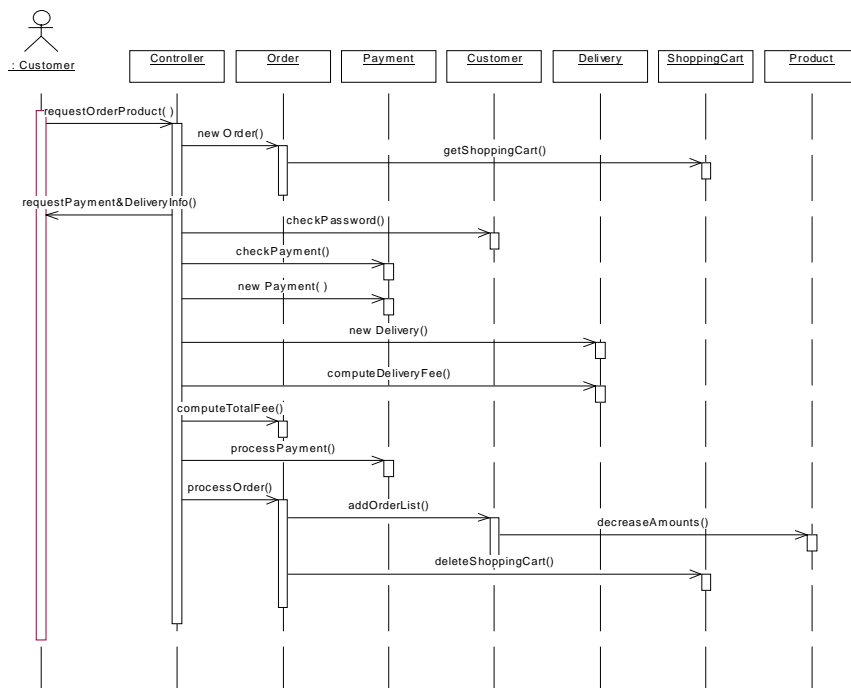


Fig. 5. Sequence diagram.

**Step 4:** Group Use Cases participating in *include* and *extend* relationships. Proceed with further grouping considering similar functionality from the perspective of a system service, the degree of usage of the identical set of objects, and actors.

Conduct an analysis focusing on Customer, Shopping Cart, Product, Order Payment, and Delivery in the class diagram shown in Fig. 4. Although Use Cases (Table 3) processing Shopping Cart reference the Shopping Cart and Product classes, they should be part of the order management system because the Use Cases reflect functionality for making orders. Therefore, the identified candidate components are member management, order management, and product management as shown in Fig. 4. Carefully examining the classes that are members of the identified candidate components reveals that the system components share the Customer and Product classes. That is, the member management and product management system components share the Customer class. The order management and product management system components share the Product class. As a result, the identified system components and their classes are as shown in Table 4.

**Table 3. Use Case diagram and class relationship table.**

System Component	UseCase	customer	Order	Payment	Delivery	Shopping Cart	Product
Member Management	Join Membership	V					
	Delete Membership	V					
	Update Membership	V					
	Search Membership	V					
Order Management	Order Product	V	V	V	V	V	V
	Cancel Order	V	V	V	V	V	V
	Update Order	V	V	V	V	V	V
	Search Order	V	V	V	V		
	Search Delivery	V	V	V	V		
	Deliver Order	V	V		V		
	Update Deliver	V	V	V	V		
	Cancel Deliver	V	V	V	V		
	Return Product	V	V				V
	View ShoppingCart					V	
	Delete ShoppingCart					V	
	Update ShoppingCart					V	V
	Add Item to Shopping					V	V
Product Management	Add Product						V
	Update Product						V
	Delete Product						V
	View Product						V
	Search Product						V

**Table 4. Identified candidate system components and their classes.**

System Component	Contained Classes
Member Management	Customer
Order Management	Customer, Order, Payment, Delivery, Shopping Cart, Product
Product Management	Product

**Table 5. UseCases and their referenced classes in terms of CRWD relationships.**

System Component	Use Case	Customer	Order	Payment	Delivery	Shopping Cart	Product
Member Management	Join Membership	C					
	Delete Membership	D					
	Update Membership	W					
	Search Membership	R					
Order Management	Order Product	R	C	C	C	R	W
	Cancel Order	R	D	D	D	D	W
	Update Order	R	W	W	W	W	W
	Search Order	R	R	R	R		
	Search Delivery	R	R	R	R		
	Deliver Order	R	R		W		
	Update Deliver	R	W	W	W		
	Cancel Deliver	R	W	W	D		
	Return Product	R	W				W
	View ShoppingCart					R	
	Delete ShoppingCart					D	
	Update ShoppingCart					W	R
	Add Item to Shopping					C	R
Product Management	Add Product						C
	Update Product						W
	Delete Product						D
	View Product						R
	Search Product						R

**Step 5:** One must determine what system component must own common classes shared by system components. Therefore, the common class reallocation decides what system component owns the Customer and Product classes using the application rules in subsection 3.3. The class relationships and CRWD matrix showing the relationships between Use Cases and classes indicate that rule 4 is applicable in this case (not 1, 2, 3, and 5 in Table 1). Customer belongs to member management, and Product belongs to product management. This is due to the fact that Customer is created by the membership system component while Product is created by the product management system component.

Therefore, the determined system components and their classes are shown in Table 6.

**Table 6. Identified system components and their classes.**

System Component	Re-arranged Classes
Member Management	Customer
Order Management	Order, Payment, Delivery, Shopping Cart
Product Management	Product

Fig. 6 is a digram showing the identified system components and their classes.

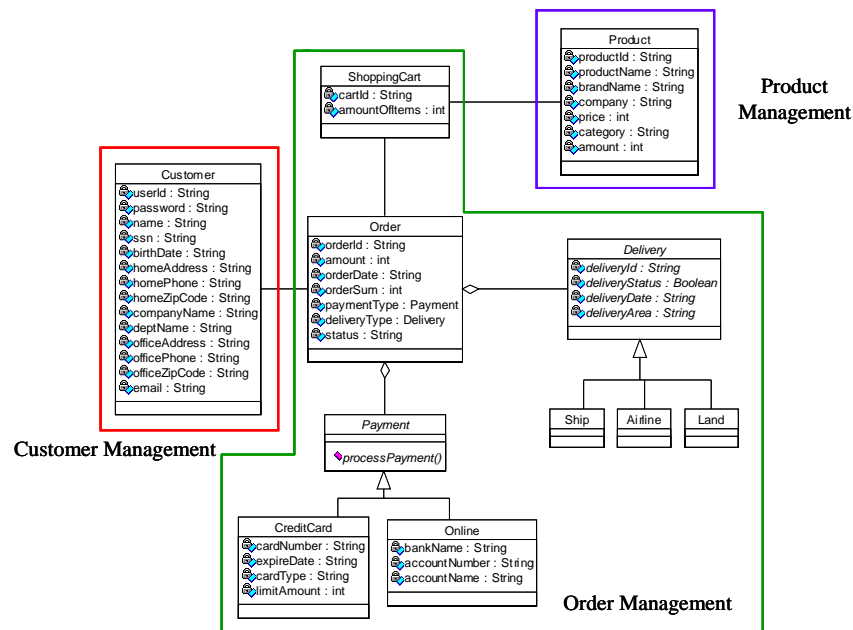


Fig. 6. Identified system components.

**Step 6:** Identify business components using the identified system components. In the case of the member and product management system components, there is no need for identifying a business component because the system components contain only one class each, that is, in itself, a business component. However, we still need to identify business components out of the order management system component. Sections 3.1 and 3.2 stated that classes found in composition (aggregation) and inheritance relationships can be grouped into a single class. Therefore, one can group Payment and Delivery into a single class and refer to it as Order since Order contains Payment and Delivery (Order = Order + Payment + Delivery). Next, we must consider Order and Shopping Cart.

**Step 7:** Sections 3.1 and 3.2 stated that one needs to define a message call relationship graph between classes inside system components using a sequence diagram in order to identify business components. Fig. 7 shows the message call relationship graph derived from the sequence diagram in Fig. 5. Note that Order (Order + Payment + Delivery) and Shopping Cart are the only classes to consider for identifying business components.

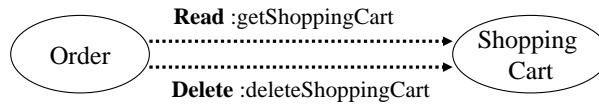


Fig. 7. Message call relationships between classes.

**Step 8~9:** Sections 3.1 and 3.2 stated that one must group classes participating in create and delete relationships into one class and define it as a candidate business component.

According to the message call relationship graph in step 7, Order and Shopping Cart participate in a delete relationship and are therefore grouped into a candidate business component. There is no further room for more grouping in this candidate business component. Therefore, the candidate business component is qualified as a full-blown business component without any more scrutiny. The system components and business components identified using the steps 1 through 9 are presented in Table 7.

**Table 7. Identified system and business components and their classes.**

System Component	Business Component	Contained Classes
Member Management	Customer	Customer
Order Management	Order	Order, Payment, Delivery, Shopping Cart
Product Management	Product	Product

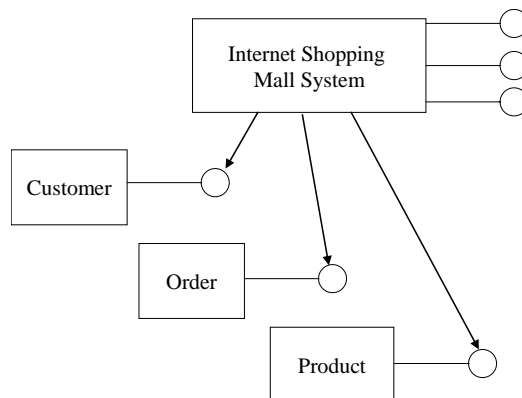


Fig. 8. System component architecture.

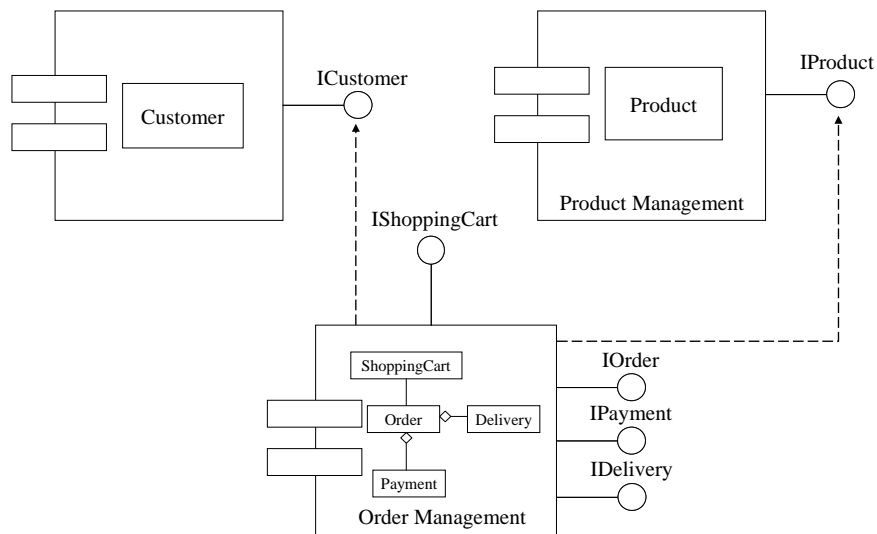


Fig. 9. Component interfaces.

Therefore, we have shown that one can identify system components (which are the units of a subsystem) and naturally identify business components that are meant to be included in a system component. For the development of each subsystem, one just needs to use the already identified components as they are. When a new system needs to be developed, one can assemble the identified business components to build it. Figs. 8 and 9 show the system component architecture and its component interfaces for the electronic commerce system built upon the identified business components.

## 5.2 Analysis and Evaluation

To prove the effectiveness of our approach identifying business components from their corresponding system components, we evaluate the coupling [11, 12] of a system for cases when CRWD factors are applied and not applied.

We measure coupling by:

- assigning different weights to the message call types in the order of create, delete > write > read.
- considering the direction and number of message calls between classes in a sequence diagram since message call types decide the degree of dependency for classes participating in a peer relationship.

Table 8 shows message types between classes, the number of messages, the direction of method calls.

In the table, message call directions are represented in a from-a-row-to-a-column fashion. Each cell contains a message call type followed by the number of messages surrounded by parentheses. For example, row 3 and column 2 of Table 8 shows that the

**Table 8. Message call types and the number of messages.**

Column Class \ Row Class	Customer (A)	Order (B)	Shopping Cart (C)	Product (D)
Customer (A)				
Order (B)	Read (9)		Read (1), Delete (1)	Write (3)
Shopping Cart (C)		Write (1)		Read (2)
Product (D)				

**Table 9. Coupling without applying weights.**

Class	$A \leftrightarrow B$	$B \leftrightarrow C$	$C \leftrightarrow D$	$B \leftrightarrow D$
# of Message Call	9	3	2	3
Coupling	9	3	2	3

**Table 10. Coupling when applying weights.**

Class	$A \leftrightarrow B$	$B \leftrightarrow C$	$C \leftrightarrow D$	$B \leftrightarrow D$
# of Message Call	9	3	2	3
Coupling	9	24	2	9

We applied weights in the order of: Create, Delete > Write > Read.

direction of a message call as Order  $\rightarrow$  Customer and that the message call type is Read. The number of messages is 9.

Table 9 shows coupling by only considering the method call direction and number without applying weights. Here classes  $A$ ,  $B$ ,  $C$ , and  $D$  respectively represent Customer, Order, Shopping Cart, and Product.

Table 9 shows that coupling between classes  $A$  and  $B$  are the highest. Therefore, it makes sense to identify a business component combining classes  $A$  and  $B$ . However, this result does not match what our intuition indicates. Table 10 shows coupling when assigning weights based on the message call direction, number, and method call types.

Table 10 reveals that coupling between classes  $B$  and  $C$  is the highest although the number of method calls is relatively smaller between classes  $B$  and  $C$ . This is because the degree of dependency is now reflected as weights. Therefore, we conclude that classes  $B$  and  $C$  must be grouped and identified as a single business component. This result matches what our intuition indicates. Next we develop all the possible candidate system components by first defining business components and combining them.

Table 11 shows the average coupling of results presented in Tables 9 and 10 when focusing on derived system components.

$S1 (B1, \dots, Bn)$  stands for candidate system component 1 (business component 1, ..., business component  $n$ ). We define the Average Coupling of a System Component as the summation of Coupling between Business Components/the number of business components. Let  $CBC_x (x = 1 \dots u)$  be the coupling among business components,  $BC_i (i = 1 \dots m)$ ,  $BC_j (j = 1 \dots n)$ . The equation to calculate ACSC can be defined as:

$$ACSC(S_p) = \frac{\sum_{x=1}^u CBC_x(BC_i, BC_j)}{u}.$$

**Table 11. Average coupling of candidate system components.**

Candidate System Component	Without Weight : Average Coupling	Applying Weight : Average Coupling
<i>S1 (A, B, C, D)</i>	4.3 (17/4)	11 (44/4)
<i>S2 (AB, C, D)</i>	2.67 (8/3)	8.67 (26/3)
<i>S3 (A, BC, D)</i>	4.67 (14/3)	6.67 (20/3)
<i>S4 (A, B, CD)</i>	5 (15/3)	14 (42/3)
<i>S5 (AB, CD)</i>	3 (6/2)	16.5 (33/2)
<i>S6 (ABC, D)</i>	2.5 (5/2)	4.5 (9/2)
<i>S7 (A, BCD)</i>	4.5 (9/2)	4.5 (9/2)

*S6* and *S7* have been already identified as business components since they were candidate components whose average coupling was the lowest when weights were applied. The same applies to *A* and *D* (each representing Customer and Product). Therefore, we just need to consider *S1* and *S3* whose business components are *A* and *D*. Order (*B*) and Shopping Cart (*C*) have a close relationship and cannot exist independently from each other. Order and Shopping Cart cannot be separated from each other because an order is generated by using the information contained in an instance of Shopping Cart. Table 11 shows that the coupling of *S1* (composed of *B* and *C*) is the lowest, which goes against our intuition. When weights are applied, *S3* (composed of *B* and *C*) has the lowest coupling, which matches our intuition. Therefore, we discovered that the existing component identification methods fail to correctly identify business components since they decide the dependency relationship between classes by only counting the number of message calls. Classes must be grouped together when they involve method call types such as *create* and *delete* despite the fact that the number of message calls between them is 1. The UDA (Usecase Data Access) table of Marmi III and the method adopted by [7] do not consider the patterns and direction of message calls and take into account only what classes are necessary to execute the functionality of Use Cases although they apply CRWD factors. Advisor, CBD 96, and Marmi III heavily depend on the experiences and intuitions of individual developers and do not provide clear standards although they claim that they identify business classes by grouping closely related boundary classes with an emphasis on core classes. System components contain business components and are executed by the interactions between business components. Therefore, it is essential to identify business components in relation to system components. However, the existing methods focus on identifying only business components in the context of the entire domain. For example, Advisor defines system components but does not provide methods and standards to identify system components. It identifies system and business components in an isolated manner. Therefore, one needs to analyze each Use Case (acting as an interface of each system component) and to decide what class belongs to what system component in order to combine business components into a system component. It is also

necessary to reconsider what is the business component that owns the corresponding class.

Our approach avoids these problems of the existing methods by identifying business components out of system components. It can also effectively identify system components through its concrete methods and standards. It can identify business components out of system components because there exists no common class shared by the identified system components. Furthermore, our approach can significantly reduce the developers' effort and time since it decomposes the entire system into system components (the unit of a system) and groups classes (participating in composition (aggregation) and inheritance relationships) into a single class, which eventually leads to the reduction of the numbers of classes to consider.

Table 12 shows the results of comparison between our approach and the existing methods in terms of the different ways available for identifying components.

**Table 12. Comparison Results between our approach and existing methods.**

Factor \ Method	CBD96	Advisor	RUP	MARMI III	[7]	Our Method
Identifying system components	1	2	2	1	2	4
Identifying business components	4	4	1	4	4	4
Identifying business components based on System components	1	1	1	1	2	4
Applying method call pattern between classes	1	1	1	1	1	4
Applying method call direction between classes	1	1	1	1	1	4
Applying structure relationship between classes	3	3	2	3	4	4
Applying relationships between use case and class	1	1	1	4	4	4
Applying method pattern between use case and class	1	1	1	4	4	4
Total	13	14	10	19	22	32

The criteria used in Table 12 are as follows:

1. No method has been used.
2. A method has been used but not defined.

3. A method has been used and defined without detailed procedures and standards.
4. A method has been used and defined with detailed procedures and standards.

## 6. CONCLUSIONS

This paper proposes a novel method to identify components that are building block for component-based systems. Our method is unique in that it does not identify business components out of the entire domain but identifies business components out of system components (as the unit of a subsystem). It addresses the deficiencies (especially, the difficulty in identifying system components) of the existing methods by providing detailed guidelines and procedures. Our guideline shows how to identify business components using the static and dynamic characteristic of relationships between classes and solves the problem of inherent dependency (of the existing methods) on the experiences and intuitions of individual developers. We have applied our method to real life systems in the areas of hospital management systems, banking systems, auction systems, and order management systems, and validated the accuracy of our approach. The results obtained by using our method matched what our intuition indicated. In the future, we plan to find more salient features of more effectively identifying components and apply them to our component development methodology.

## REFERENCES

1. D. F. Dsouza and A. C. Wills, *Objects, Component, and Frameworks with UML: the Catalysis Approach*, Addison Wesley, 1999.
2. Compuware, *About Uniface*, <http://www.compuware.com/products/uniface/about.htm>, 2001.
3. Computer Associates, *COOL:Joe 2.0*, [http://www.cai.com/products/cool/joe/cooljoe\\_pd.pdf](http://www.cai.com/products/cool/joe/cooljoe_pd.pdf), 2001.
4. J. Cheesman and J. Daniels, *UML Components: A Simple Process for Specifying Component-Based Software*, Addison Wesley, 2001.
5. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison Wesley, 1999.
6. ETRI, "Component development methodology – MARMI- III," Technical Report, Electronics Telecommunication Research Institute, 2002.
7. J. K. Lee, S. J. Jung, and S. D. Kim, "Component identification method with coupling and cohesion," in *Proceedings of Asia Pacific Software Engineering Conference*, 2001, pp. 79-88.
8. E. S. Cho, S. D. Kim, and S. Y. Rhew, "A domain analysis and modeling methodology for component development," *International Journal of Software Engineering and Knowledge Engineering*, Vol. 14, 2004, pp. 221-254.
9. D. C. Kung, J. Gao, P. Hsia, F. Wem, Y. Toyoshima, and C. Chen, "Change impact identification in object oriented software maintenance," in *Proceedings of the International Conference on Software Maintenance*, 1994, pp. 202-211.
10. D. C. Kung, J. Gao, and P. Hsia, "Class firewall, test order, and regression testing of

object-oriented programs,” *Journal of Object-Oriented Programming*, Vol. 8, 1995, pp. 51-65.

11. B. Henderson-Sellers, *Object-Oriented Metrics*, Prentice-Hall, 1996
12. S. R. Chidamber and C. F. Kemerer, “A metric suite for object-oriented design,” *IEEE Transactions on Software Engineering*, Vol. 17, 1994, pp. 636-638.



**Misook Choi** received her B.E. degree from the Chonbuk University at Jeonju of Korea in 1990. She received both M.S. degree and Ph.D. degree in Computer Science from the Sookmyung Woman’s University at Seoul of Korea in 1994 and 2002, respectively. She worked as a full-time instructor in the department of Software Development at Naju College from 1995 to 1999. She is a full-time instructor in the Department of Computer Engineering at Woosuk University. Dr. Choi’s research focus is software engineering, object-oriented modeling technique, CBSE, MDA, software reuse, service-oriented architecture, and software metrics.



**Eunsook Cho** received her B.E. degree in Computer Science from the Dongeui University at Busan of Korea in 1993. She received both M.S. degree and Ph.D. degree in Computer Science from the Soongsil University at Seoul of Korea in 1996 and 2000, respectively. She worked as a invited researcher in ETRI (Electronics Telecommunication Research Institute) at Daejeon of Korea from 2002 to 2003. She worked as a full-time instructor in the Department of Computer Science at Dongduk Women’s University from 2000 to 2004. Dr. Cho is a full-time instructor in the Department of Software at Seoil College. Dr. Cho’s research focus is software engineering, object-oriented modeling technique, software architecture, CBSE, MDA, software reuse, and ubiquitous and embedded computing.