A Method and Tool for Identifying Domain Components Using Object Usage Information

Woo-Jin Lee, Oh-Cheon Kwon, Min-Jung Kim, and Gyu-Sang Shin

To enhance the productivity of software development and accelerate time to market, software developers have recently paid more attention to a component-based development (CBD) approach due to the benefits of component reuse. Among CBD processes, the identification of reusable components is a key but difficult process. Currently, component identification depends mainly on the intuition and experience of domain experts. In addition, there are few systematic methods or tools for component identification that enable domain experts to identify reusable components. This paper presents a systematic method and its tool called a *component identifier* that identifies software components by using object-oriented domain information, namely, use case models, domain object models, and sequence diagrams. To illustrate our method, we use the component identifier to identify candidates of reusable components from the object-oriented domain models of a banking system. The component identifier enables domain experts to easily identify reusable components by assisting and automating identification processes in an earlier development phase.

Manuscript received June 7, 2002; revised Dec. 6, 2002.

This work was supported by the National Research Laboratory (NRL) Program of the Ministry of Science and Technology of Korea.

Woo-Jin Lee (phone: +82 53 950 6378, e-mail: woojin@knu.ac.kr) is with the Department of Computer Science, Kyungpook National University, Daegu, Korea.

Oh-Cheon Kwon (phone: +82 42 860 1144, e-mail: ockwon@etri.re.kr), Min-Jung Kim (e-mail: minjkim@etri.re.kr), and Gyu-Sang Shin (e-mail: gsshin@etri.re.kr) are with Computer & Software Research Laboratory, ETRI, Daejeon, Korea.

I. INTRODUCTION

In the rapidly evolving and enlarging software market, there is a great need for enhancing the productivity of software development and accelerating time to market. As a solution for this problem, the component-based development (CBD) method has recently been introduced into software development organizations, because the method supports parallel or incremental development, plug-and-play features, and easy component reuse and maintenance. In general, a software component is specified as a collection of objects. The software component can be independently developed, delivered, and composed with other components without modifying the source code and has explicit and well-specified interfaces [1]. The reusability of components is one of the factors that have made the CBD method successful. Among the CBD processes [1], the process of identifying reusable components is the most important and difficult process because it is performed in terms of domain properties or business logics and it does not include design or implementation issues in an earlier development phase.

Since only a few systematic identification methods have been designed, there are no automatic or semi-automatic tools that enable domain experts to easily and efficiently identify reusable components by guiding or assisting identification processes in an earlier development phase. Component identification is mainly performed by domain experts' intuitive procedures or experiences without systematic tools. The rational unified process (RUP) [2] provides, in three different perspectives, intuitive or human oriented identification methods based on process design, object dependency, and subsystems at the architecture level. Choi and et al. [3] provided an enhanced version of the RUP approach, in which service components were defined as objects extracted from closely related use cases and in the refinement step, business components were defined as common objects that appear in several service components. In major CBD tools, such as TogetherSoft's Together [4], Computer Associates' Cool:Joe [5], and Compuware's Uniface [6], the component identification process is not systematically supported. Together and Uniface do not support the component identification process, while Cool:Joe supports the identification process by grouping closely related objects with core types, which are specified by users.

In this paper, we propose a systematic method and its supporting tool for identifying software components from object-oriented domain models, namely, use case diagrams, class diagrams, and sequence diagrams (Fig. 1). These object-oriented domain models can be obtained from a domain analysis process, in which common domain objects and common use cases are extracted through commonality and variability analysis. Assuming that common class diagrams, common use cases, and sequence diagrams are given after the domain analysis process, we focus on the component identification process, in which we clearly define dependencies among objects and propose object clustering algorithms.

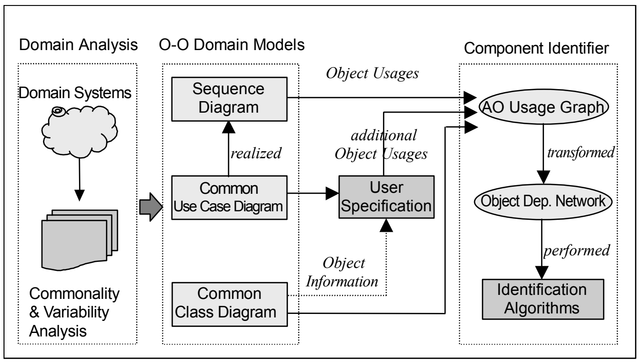


Fig. 1. Overview of the component identification procedure.

We describe the object-oriented domain models from three viewpoints—structural, functional, and behavioral—along with their corresponding domain models —class diagrams, use case diagrams, and sequence diagrams. To precisely describe the dependencies among objects, we merge these three viewpoints into a uniform model, in which we extract the structural relationships among objects from class diagrams. To clarify ambiguous dependencies among objects, we extracted object usages, which represent usage relationships among objects such as create, destroy, update, and reference, from sequence diagrams automatically or additionally specified the object usages according to use cases. We weighted each object usage according to the frequency or significance of each use case.

For uniformly describing object usages and structural dependences in a single notation, we propose an *actor and object usage graph* (AO usage graph). To perform the clustering algorithms, we provide a new graph concept, called the *object dependency network*. An object dependency network can be obtained from the AO usage graph by calculating a weighted value for the accumulated object usages and by eliminating actor nodes. We provide on the basis of the object dependency network, two object clustering algorithms called a seed algorithm and a cohesion algorithm. In addition, we provide a support tool called the *component identifier*, which was integrated into the Component-Based Application deveLopment Tool (COBALT) [7] developed by ETRI. Using the integrated tool, we carried out a case study for developing a simplified Internet-based banking system in order to adjust the weight values of dependency types and to check the applicability of the clustering algorithms.

The rest of the paper is structured as follows: section 2 briefly describes the component development process. In section 3, we describe the design of the component identifier. Section 4 gives the definitions for the AO usage graph and object dependency network for describing object dependencies in formal notation. In section 5, the component identification algorithms are explained step by step for a banking system example. Section 6 specifies the component identifier tool. Finally, in section 7, we conclude our work and provide an outlook on future work.

II. COMPONENT DEVELOPMENT PROCESS

The CBD process is classified into two major processes: the component development (CD) process, which builds reusable components, and the component-based software development (CBSD) process, which assembles pre-built components into an application system. In this study we focused on the CD process. Before we explain the CD process, we will describe the definition and properties of a software component. There are several definitions of a component. D'Souza et al. [1] defined a component as "an independently deliverable unit of software that encapsulates its design and implementation and offers interfaces to the outside, by which it may be composed with other components to form a larger whole." In addition, the Object Management Group defined a component in the Unified Modeling Language Specification [8] as "a distributable piece of implementation of a system, including software code (source, binary or executable) but also including documents, etc."

The following properties of components are generally accepted.

- Encapsulated: a component hides its implementation or code that drives a component. The consumer of a component can access the component using its interfaces.

- Descriptive: a component must publish information about itself including its interfaces, implementations, and deployment conditions.
- Replaceable: implementation details of a component can be changed without affecting the consumers of the component and can be provided if there is no change in the component interface.
- Extensible: it is possible to enlarge or extend its range of services without affecting the consumers, delegating responsibility, or adding interfaces.

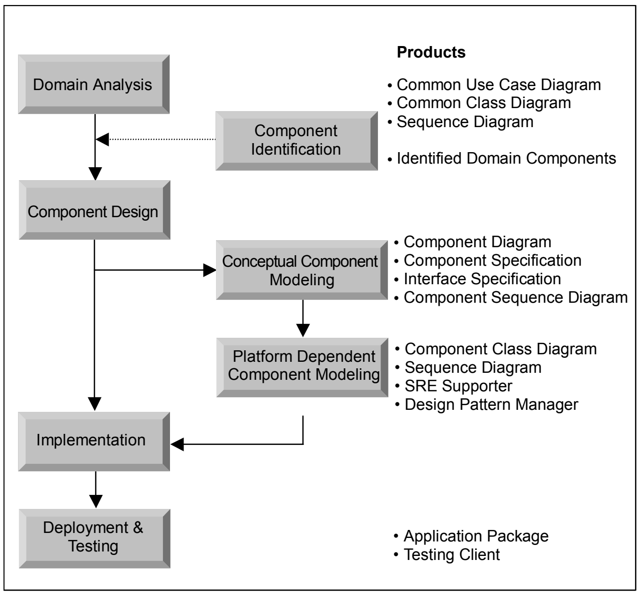


Fig. 2. Component development process.

In order to develop highly adaptable and reusable components, we propose our CD process model as depicted in Fig. 2. Unified Modeling Language (UML) [9], which builds several diagram models, such as class diagrams, use case diagrams, sequence diagrams, collaboration diagrams, and so on, is widely used in describing domain models in the software industry. Our approach describes the output of domain modeling in UML diagram notation. In addition, we divide the component modeling into two phases: conceptual component modeling and platform-dependent modeling. In the perspective of domain characteristics, conceptual component modeling identifies conceptual components as reusable units without considering platform constraints and implementation details. In this step, component interfaces are clearly defined and their dependencies described. In platform-dependent modeling, each conceptual component is internally designed by using platform-specific properties, such as entity beans and session beans. By using a two-layered component modeling approach, reusability of components may be further improved. Since conceptual components and their interfaces are specified in a conceptual modeling process, they are more reusable and can be reused for any system. If someone wants to port a component to another component framework without affecting the component's functionalities, he or she rebuilds a new platform-specific component based on the same conceptual component.

- Domain Analysis: Domain analysis is an activity for identifying objects and operations in a set of related systems and for identifying common objects among the set of existing systems through commonality analysis and variability analysis. Domain analysis plays a key role in the success of finding reusable domain components because it can efficiently analyze a specific domain and sufficiently provide its characteristics.
- Component Identification: Component identification is an activity for identifying components from various domain models. In an intuitive manner, users can manually identify components from domain models. Otherwise, components are systematically identified by using identification algorithms based on object-relationship factors and object usages.
- Component Design: Component design is an activity for modeling each component according to its specification. Component design consists of the two modeling processes, conceptual component modeling and platform-dependent modeling. Assuming that component interfaces are not changed, the detailed design of each component can be performed independently according to its specification.
- Implementation: Implementation is an activity for adding business logics including the details of how the operations of an interface will work.
- Deployment & Testing: Deployment & testing is an activity for packaging and deploying components to an application server and for checking the interface functionality of a deployed component.

Although the CBD process has many advantages, some considerations should be taken into account when applying the CBD process to real projects. First, at the earlier stage, the CBD process requires much cost and effort because of learning curves. Second, the most important factor for successfully applying the CBD process is mainly dependent on the number or quality of existing reusable components. In order to enhance the reusability of components, we focus on identifying highly cohesive domain components with low coupling. Other processes, such as domain analysis, component design, implementation, and deployment & testing, are performed with the COBALT:Constructor tool [7].

III. STRUCTURE OF THE COMPONENT IDENTIFIER

The component identifier plays the role of partitioning a large system into manageable components (or subsystems) by analyzing its domain models. Input data, such as use case diagrams, class diagrams, and sequence diagrams, are obtained from the Domain Modeler as shown in Fig. 3. The identified components are provided to the Component Modeler for performing further steps of component development, such as interface definition, component dependency description, component design, and component implementation.

As Fig. 3 illustrates, the component identifier is composed of the Usage-Management Wizard, Algorithm-Performing Wizard, Domain Model Translator, Object Dependency Network Generator, and Object Clustering Engine.

The Usage-Management Wizard receives the structural dependency among objects from the class diagram and the object usages that may be extracted from sequence diagrams or may be additionally provided by users. The Object Dependency Network Generator calculates weight values between objects by considering the structural dependency and usage relationships of objects. It also generates an object dependency network by representing object dependencies as weighted arcs. The Algorithm-Performing Wizard receives threshold values, such as a clustering threshold (CT) and a seed object threshold (SOT), from users. Based on clustering criteria such as CT and SOT, the Object Clustering Engine performs clustering algorithms for grouping closely related objects.

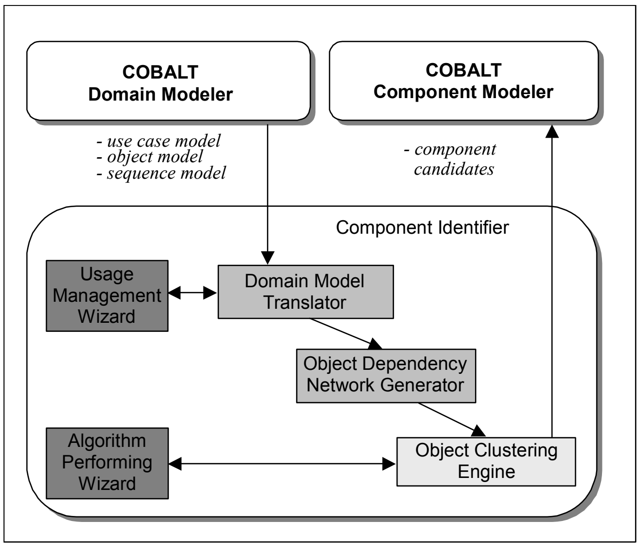


Fig. 3. Structure of the component identifier.

IV. REPRESENTATION OF OBJECT RELATIONSHIPS

In order to support a systematic identification procedure, domain modeling information should be precisely described by formal notations. In this section, the actor and object usage graph (AO usage graph) and object dependency network are provided as formal notations for describing domain models.

1. Actor and Object Usage Graph

Object-Oriented (O-O) domain modeling may be mainly performed from the perspective of structural, functional, and behavioral views. Object modeling finds and defines core domain objects that perform major roles in structuring a system. Use case diagrams and sequence diagrams are used to describe functional requirements and the behavior of a system, respectively. To specify the correlation of the different domain models in a uniform style, it is necessary to merge the domain models into a single formal model.

As an example, we provide an Internet-based banking system with functionalities such as deposit management, customer management, customer authentication, journaling, and bookkeeping. Figure 4 shows a class diagram including common domain objects involved in the banking system. For simplicity, the dependency relationships, class attributes, and methods are omitted in Fig. 4.

In Fig. 4, a customer, such as a private or corporation customer, may have zero or multiple accounts. Customers can change customer information via transactions of a customer (CustomerTX) and can perform ordinary transactions (OrdinaryTX), such as opening an account, depositing, withdrawing, and transferring money. A single transaction is related to a set of accounts. When performing a transaction, a set of journaling and bookkeeping information can be recorded via Journaling and BookKeeping classes, respectively.

A sequence diagram can be viewed as one realization of a use case. Figure 5 shows two sequence diagrams that realize the “register a private customer” and “create a new account” use cases, respectively. In Fig. 5(a), a private customer registers his/her personal record through the interfacing class CustomerTX, which is inherited from the Transaction (TX) class. The personal record is stored in the PrivateCustomer class. Journaling information is recorded in the CustomerJournal class before and after important transactions.

Among the structural relationships between objects, such as generalization, composition, association, and dependency, the first two relationships of a class diagram show concrete dependencies between objects, while the other relationships are relatively ambiguous. In our approach, the association and dependency relationships are further clarified by adding

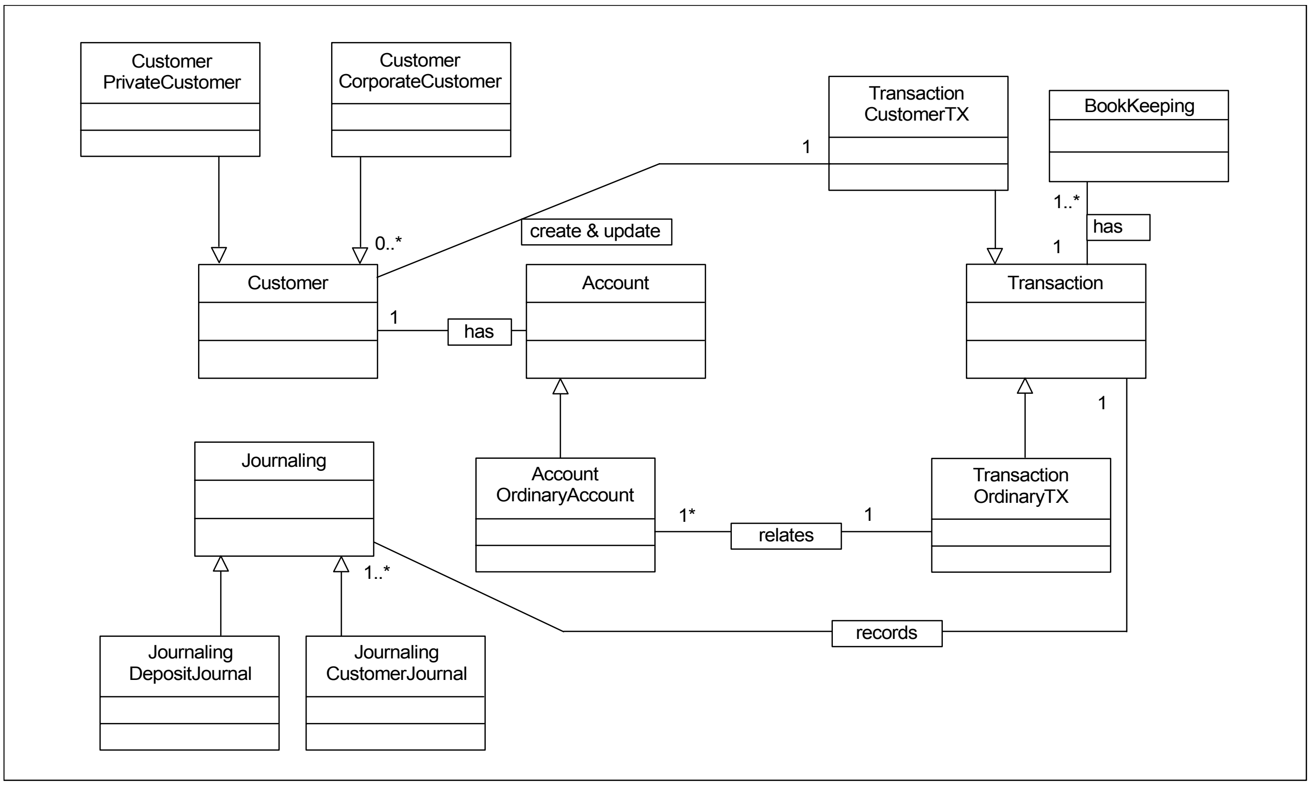


Fig. 4. A class diagram for common domain objects in a banking system.

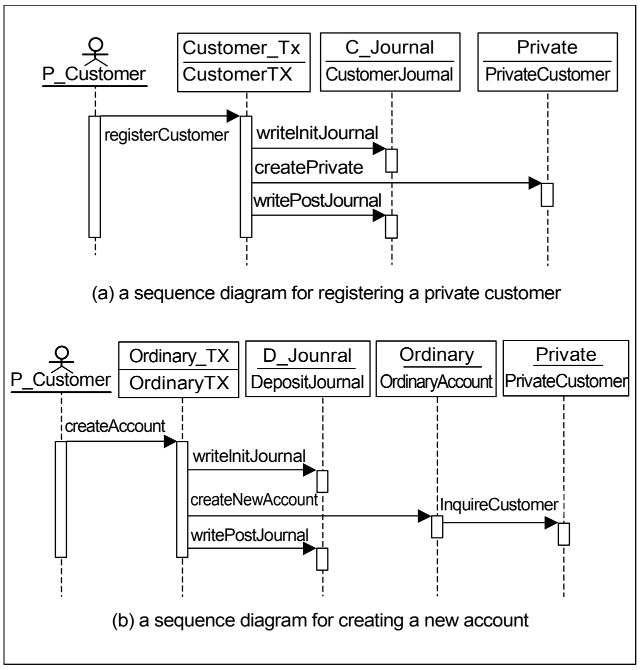


Fig. 5. Examples of sequence diagrams.

behavioral information that can be found in use cases and sequence diagrams. Thus, object usages of use cases or sequence diagrams are complemented for the association and dependency relationships. For more accurately representing domain models, the weighted value of each object usage is also specified by considering the usage frequency or the importance degree of its role. For describing the structural and behavioral relationships among objects in a uniform notation, we propose the AO usage graph. It represents the dependency of objects, such as generalization, composition, accumulated object usages, and inherited weights of object usages, from the importance degrees of use cases. The AO usage graph is defined as follows.

Definition 1. Actor and Object (AO) Usage Graph
A directed graph G = (V, E), where

- V = A ∪ O: A represents a set of actors and O represents a set of objects,
- E: a set of edges which have labels, such as generalization and composition, and a bag of weighted labels, such as create-destroy, create, destroy, update, and reference.

Although the usage patterns between actors and objects have no direct effect on the dependencies among objects, they are described in the AO usage graph because an actor plays the important role of initiating a flow of messages in a sequence diagram. The usage pattern from actors to objects can be used for determining important objects. However, dependencies from objects to actors that represent users, other systems, or hardware are not specified since they have no effect in the internal system behavior. In the AO usage graph, actors are denoted as dotted circles while objects are denoted as lined circles. The dependency relationships among objects or between actors and objects are described by seven dependency keywords as follows: generalization, composition, create-destroy, create, destroy, update**,** and reference. Each object usage has a weighted value inherited from the weight of the corresponding use case. The dependency relationship between nodes is represented by a sum of weighted object usages or a structural relationship such as generalization and composition.

Figure 6 is an example of the AO usage graph obtained from the class diagram shown in Fig. 4, and from two sequence diagrams shown in Fig. 5. The nodes and "Gen" represent the classes and generalization relationship in the class diagram, respectively (Fig. 4). Object usages, such as 1.0**Update* and 0.8**Update*+0.8**Update*, are extracted from sequence diagrams (Fig. 5) on the assumption that the weights of the use case "create a new account" and the use case "register a private customer" are 1.0 and 0.8, respectively.

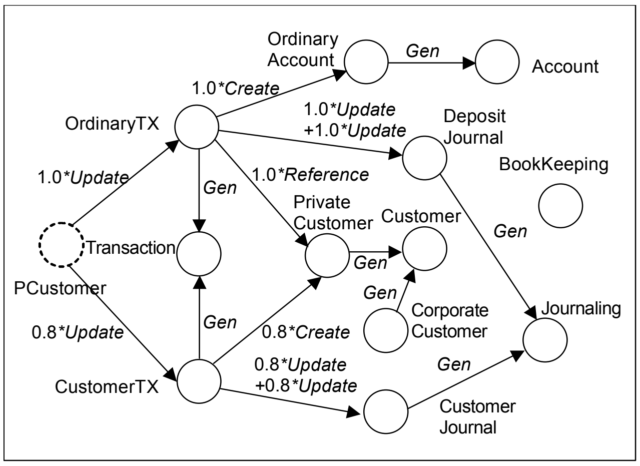


Fig. 6. An example of the AO usage graph.

2. Object Dependency Network

Although the AO usage graph is enough to describe the structural and behavioral features of the domain models in a uniform notation, it is not suitable for performing the clustering algorithm since it is difficult to compare accumulated usages attached on dependency edges. Therefore, we provide a notation, the object dependency network, in which the accumulated usages are translated into a weighted value. In addition, the importance degree of each object is calculated by adding weighted usages for each incoming arc except structural dependencies. The object dependency network is defined as follows.

Definition 2. Object Dependency Network
A directed graph G = (V, E, w), where

- V = O: O is a set of objects,
- E: a set of directed edges which have a dependency degree (a real number),
- w(v): a weight function of defining the importance degree of each vertex v.

The object dependency network represents the dependency degrees (DDs) among objects and the importance degree (ID) of each object. In the object dependency network, actor nodes and their connected arcs are transformed into self-loops with weighted values. Figure 7 shows an example of the object dependency network transformed from the AO usage graph shown in Fig. 6.

As described in Fig. 7, the dependency degrees among objects have normalized real values ranging from 0.0 to 1.0, which are obtained by summarizing the weight values of accumulated usages and being normalized according to the largest values. The value of each object usage is calculated by referencing a weight-mapping table, as shown in Table 1, which assigns a weight value to each dependency type according to the coupling strength between objects. One notable point in Table 1 is that the generalization relationship does not appear. The generalization relationship is the strongest coupling among objects since an inherited class must include its parent classes. Therefore, we specially treat the generalization relationship at the end of the identification process.

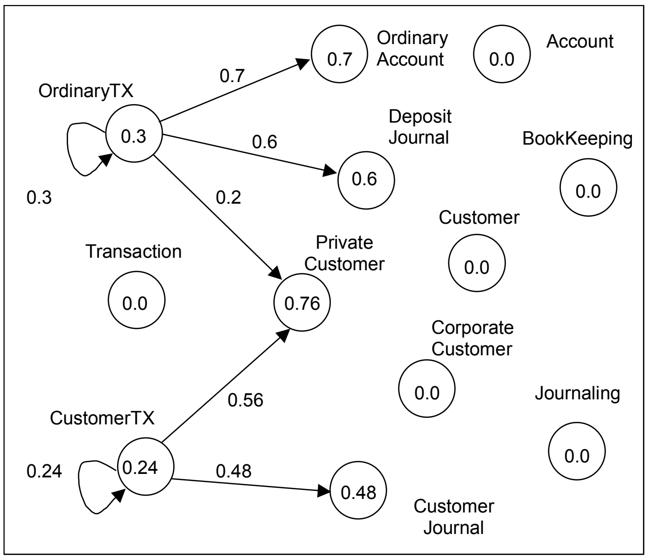


Fig. 7. An example of the object dependency network.

Strong coupling relationships, such as composition and create/destroy, have high values, while weak coupling relationships, such as update and reference, have relatively low values. In order to provide more flexibility or to reflect domain experts' experience in assigning the weight of each type, the type weights shown in Table 1 can be customized by domain experts.

Table 1. Weighted values of dependency types.

Dependency Type (DT)	Weight
Composition	1.0
Create/Destroy	0.8
Create	0.7
Destroy	0.6
Update	0.3
Reference	0.2

The DD is characterized by a weight of structural dependency or accumulated object usages. The weight of accumulated object usages is calculated by (1), that is, the DD value between two objects is obtained by summing up the weights of all the instances of object usages for each use case, where the weight of each object usage is calculated by multiplying the weights of the dependency type and the corresponding use case.

$$\sum_{i=1}^{\#\ of\ UseCase} \sum_{j=1}^{\#\ of\ InstDT} W(UseCase_i) * Weight(InstDT_j) . \qquad (1)$$

The ID value of an object is defined by adding the DD values of its incoming arcs, which may come from actors or other objects in the AO dependency graph. The ID values are used to determine important objects and to calculate relative dependencies between objects when performing identification algorithms. In the object dependency network, an ID value appears in the corresponding node as shown in Fig. 7.

V. COMPONENT IDENTIFICATION ALGORITHMS AND THEIR APPLICATION

To group closely related objects into a cohesive component, we propose the seed algorithm and cohesion algorithm. The seed algorithm performs object clustering in the pivot of important objects while the cohesion algorithm groups the closest objects incrementally and repetitively.

1. Important Objects and Clustering Criteria

In order to identify a group of objects that perform an independent and important behavior of a system, it is necessary to find an object that has a key role in the group. Then, using the key object, its closely related objects are grouped together. In our approach, an initiation object that has the role of interacting with users is considered an important object due to the initiating users' requirements. The initiation object is connected to actors. A reuse object that is commonly used by several objects is also considered an important object in the perspective of reusability. In the object dependency network, important objects can be found on the basis of user-defined SOT values as follows.

- An important initiation object is an object which has a self-loop with a greater ID value than the SOT.
- An important reuse object is an object which has two or more incoming arcs with greater DD values than the SOT.

Assuming that the SOT value is 0.4, objects O_1 and O_5, shown in Fig. 8, are initiation objects, and object O_3 is a reuse object. These important objects are used as seed objects for clustering closely related neighbor objects.

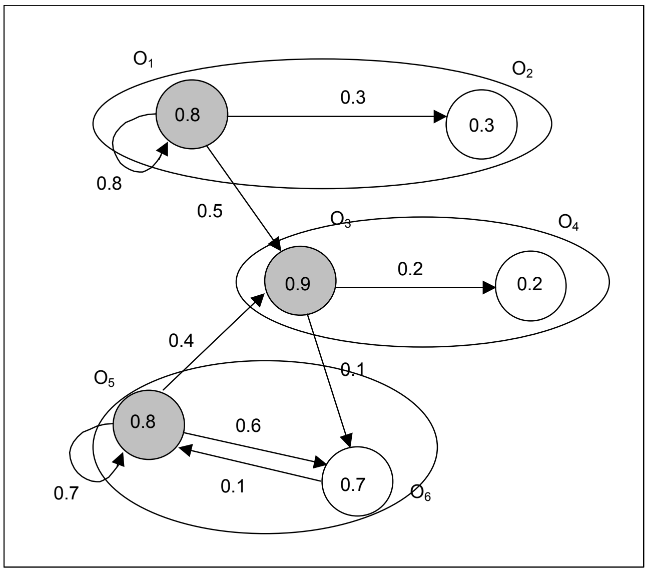


Fig. 8. Decision of important objects and object clustering.

A simple criterion for clustering a neighbor object is to check whether the DD value of a connected arc is greater than the CT. There might be an object that is only used by a single object; it is called *a dedicated object*. Since the dedicated object is only available to a calling object, it is reasonable to merge it with the calling object. However, this clustering criterion mainly fails to group the dedicated object due to its lower DD value. In order to consider dedicated objects, the clustering criterion may be slightly modified. Instead of using the DD value of an arc as clustering criteria, a relative dependency value (that is, the DD of an arc/ID of the destination arc object) is used. Since the relative dependency (RD) of the dedicated object is 1.0, it may be included in its calling object. In Fig. 8, by applying the relative dependency values to the clustering criterion, dedicated objects O_2 and O_4 can be included in their calling objects although they have lower DD values than the CT.

Dependencies among objects may be bi-directional as shown in Fig. 8. Therefore, both directional dependencies should be considered when clustering objects. Since the relative dependency means the relative importance degree among incoming arcs, it is not reasonable to add both directional dependency weights. Instead, we chose the maximum of two dependencies as the relative dependency of objects.

2. The Seed Algorithm

Using the definition of important objects and clustering criteria mentioned previously, the seed algorithm identifies cohesive components according to the procedure shown in Fig. 9.

After object information and object usages are extracted from the O-O domain models, the AO usage graph and object dependency network are generated. Then, by assigning the important objects to seed objects, objects that have a larger

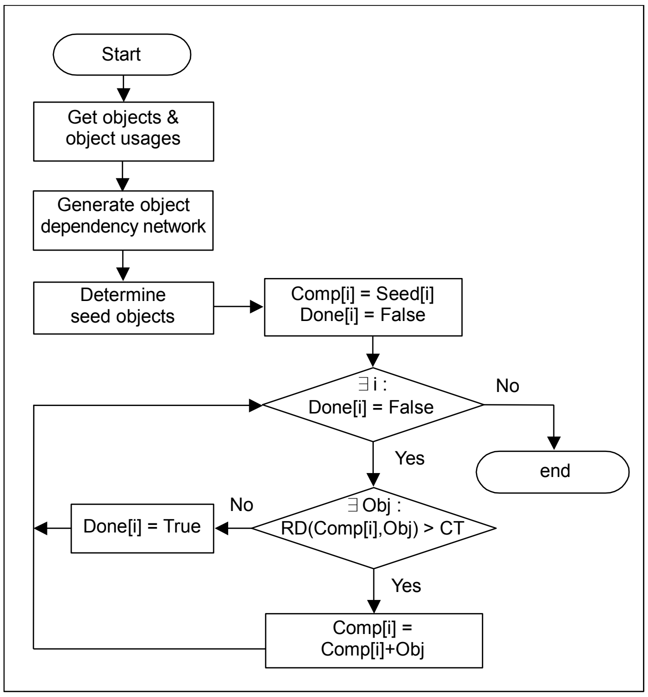

Start
Get objects & object usages
Generate object dependency network
Determine seed objects
Comp[i] = Seed[i] Done[i] = False
∃i : Done[i] = False
No
end
Yes
No
Done[i] = True
∃Obj : RD(Comp[i],Obj) > CT
Yes
Comp[i] = Comp[i]+Obj

Fig. 9. A flowchart of the seed algorithm.

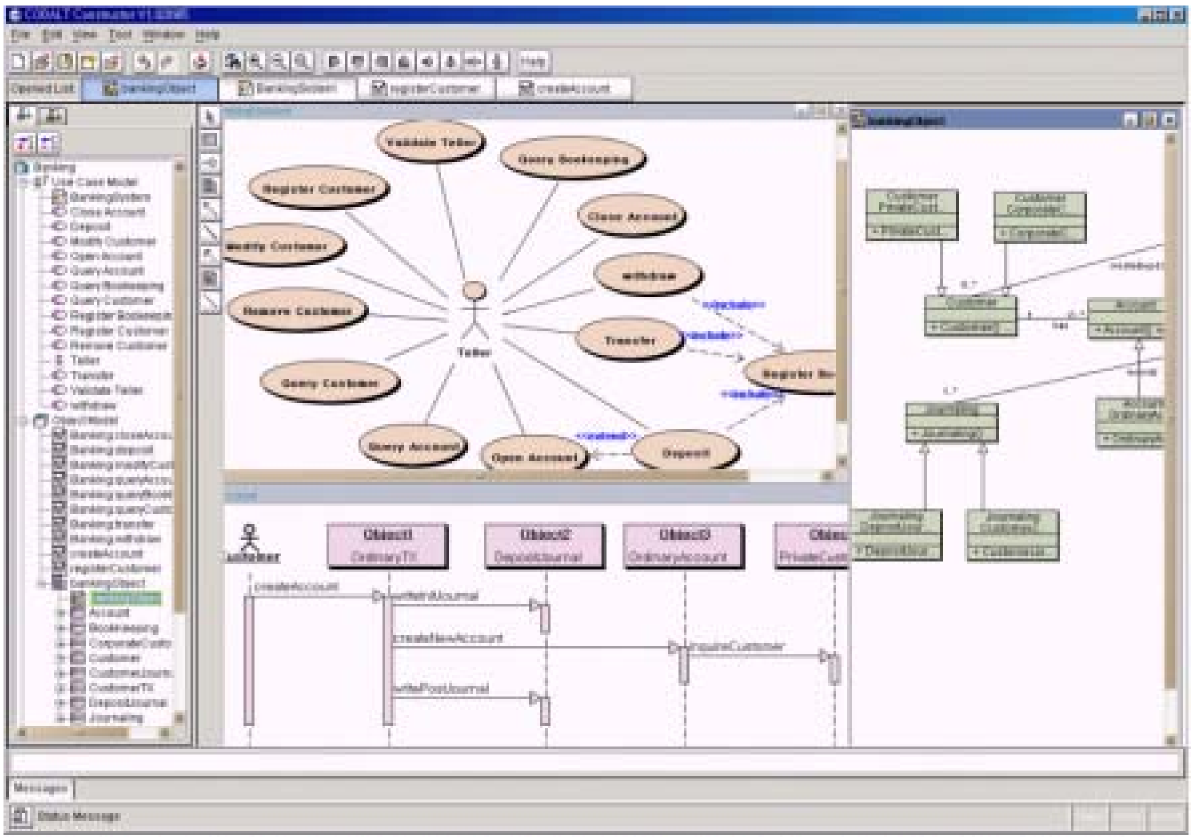


Fig. 10. O-O domain models of the banking system.

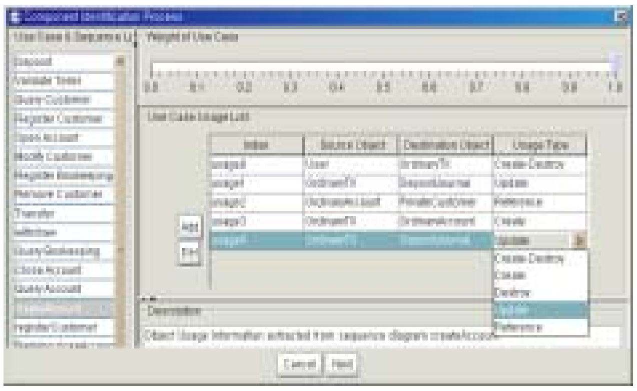
dependency value than the CT are iteratively clustered into cohesive components.

By applying the seed algorithm to the banking system step by step, we provide a more clear and illustrative explanation. As a result of a domain analysis process, the banking system is composed of 13 use cases, such as *Open Account* and *Deposit*, 18 sequence diagrams, such as *Bank-OpenAccount* and *Bank-CloseAccount*, and 12 objects, such as *Customer, Account*, and *Transaction*, as shown in the leftmost browser tree of Fig. 10. Each step of the seed algorithm is explained as follows.

- **Step 1:** To construct an AO usage graph, object information and object usages are automatically extracted from class diagrams and sequence diagrams, respectively. In addition, users can specify the importance degree of each use case or sequence diagram and can update object usages by using the Usage-Management Wizard as shown in Fig. 11. Figure 11 illustrates five object usages in the *Bank-OpenAccount* sequence diagram and its importance degree is set to 1.0.
- **Step 2:** Internally, an AO usage graph is transformed into an object dependency network by calculating each DD and ID value and by replacing actor nodes with self-loop arcs. Figure 12 shows an object dependency network of the banking system.
- **Step 3:** Using the Algorithm-Performing Wizard (Fig. 13), domain experts specify an SOT for determining seed objects. On the object dependency network shown in Fig. 12, OrdinaryTX and CustomerTX are defined as seed objects, assuming that the SOT is 0.5.

In a preparatory step for clustering objects, each seed object is assigned to a component. A condition flag, Done[i], is given to each component to determine whether further object navigation is possible. The condition flag is initialized as a 'false' value.

- **Step 4:** A terminating condition of the identification procedure is determined according to whether or not there is a component in which additional object navigation is possible. As a result of checking the terminating condition, if none of the components are allowed to perform the navigation, then the identification procedure is terminated. Otherwise, after finding a non-included object whose relative dependency is larger than a specified CT (see "Threshold for Object Clustering" sliding bar in Fig. 13), it is included into the corresponding component. This process is repeated until none of the components have further object navigation. In Fig. 12, OrdinaryAccount, DepositJournal, and BookKeeping are grouped into the seed object OrdinaryTX since their relative dependencies are 1.0. In a similar manner, PrivateCustomer, CorporationCustomer, and CustomerJournal are grouped into the seed object CustomerTX.



Index	Source Object	Destination Object	Usage Type
usage0	User	OrdinaryTX	Create-Destroy
usage1	OrdinaryTX	DepositJournal	Update
usage2	OrdinaryAccount	PrivateCustomer	Reference
usage3	OrdinaryTX	OrdinaryAccount	Create
usage4	OrdinaryTX	DepositJournal	Update

Object Usage Information extracted from sequence diagram createAccount

Fig. 11. The usage-management wizard.

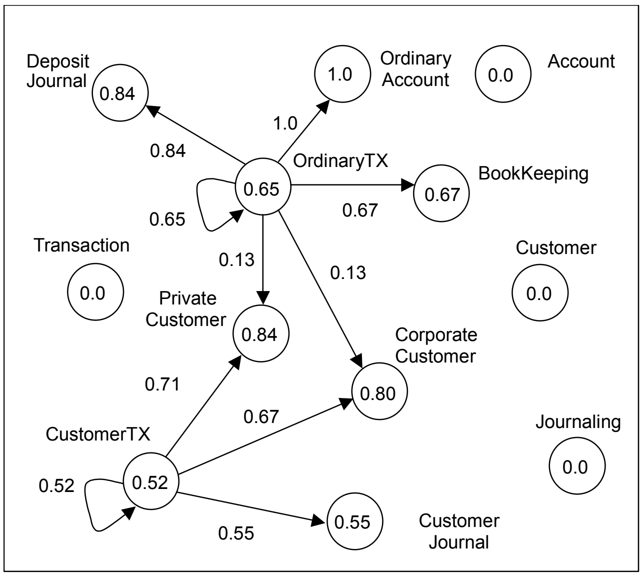


Fig. 12. An object dependency network of the banking system.

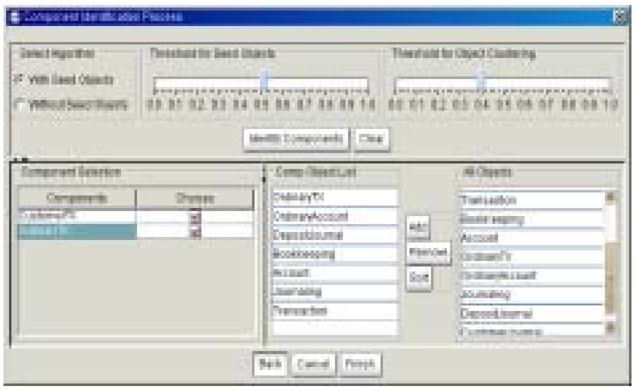


Fig. 13. The algorithm-performing wizard.

As a final step, generalization is considered. For each inherited class, its parent classes are included into the same component. Figure 14 shows the final result of the component identification process. In the banking example, two components, OrdinaryTX and CustomerTX, are identified. Each class diagram of the components is shown in Fig. 14, where the generalization relationships among classes are included.

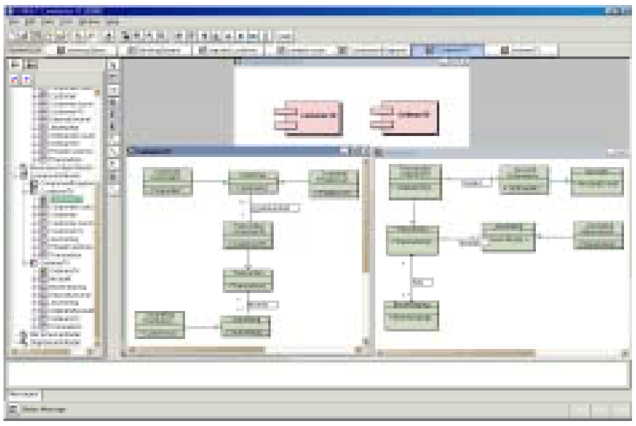
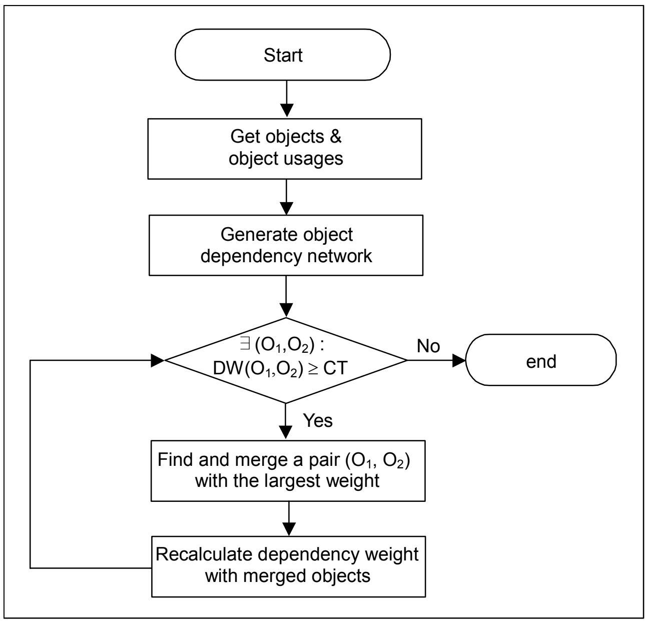


Fig. 14. An example of a component diagram.

3. The Cohesion Algorithm

The second clustering algorithm, the cohesion algorithm, which incrementally groups closely coupled objects by the criteria of the CT is shown in Fig. 15.



Start
Get objects & object usages
Generate object dependency network
$\exists (O_1,O_2)$: $DW(O_1,O_2) \geq CT$
No
end
Yes
Find and merge a pair (O_1, O_2) with the largest weight
Recalculate dependency weight with merged objects

Fig. 15. A flowchart of the cohesion algorithm.

The first two steps of the cohesion algorithm are the same as those of the seed algorithm. Figure 16 (a) shows a simplified object dependency network where IDs, self-loops, and unrelated objects are omitted since only the cohesions of objects are considered in the cohesion algorithm. The remaining steps are described as follows.

- **Step 3:** If there exists a pair of objects whose dependency weight (DW) exceeds the CT, the next clustering step is performed. Otherwise, the procedure is terminated.
- **Step 4:** Among the object pairs whose dependency exceeds the CT, the pair with the largest weighted value is selected and merged into a cluster. In Fig. 16(a), assuming that the CT is 0.5, a pair of OrdinaryAccount and OrdinaryTX is first chosen to be merged into Cluster1.
- **Step 5:** In the result of object clustering, the weight between the cluster and remaining objects should be recalculated. If an object has multiple edges with the cluster, its dependency weight is redefined by summarizing individual weights of multiple edges. After weight recalculation, the procedure proceeds to Step 3.

Figure 16 shows the iterative clustering steps of the cohesion algorithm. In Figs. 16(a), (b) and (c), three successive clusters are shown. Figure 16(c) represents the final step, where there are no more pairs of objects whose dependency exceeds CT. Thus, we found two components: one is composed of OrdinaryAccount, DepositJournal, OrdinaryTX, and BookKeeping classes; the other is composed of PrivateCustomer, CustomerTX, CorporateCustomer, and CustomerJournal. For each inherited class, its parent classes are included into the same component, as in the seed algorithm.

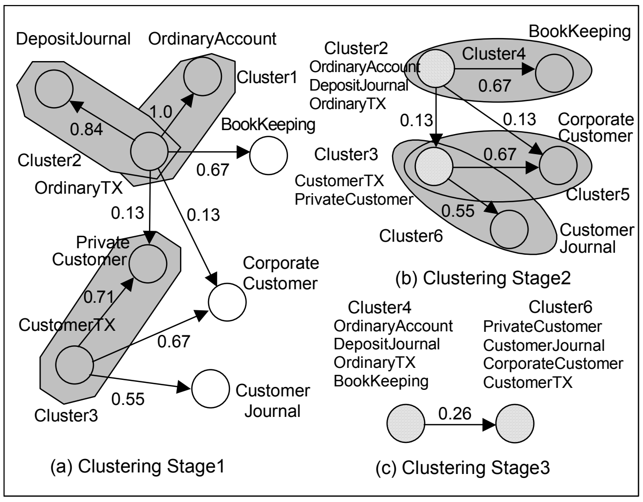


Fig. 16. Clustering steps in the cohesion algorithm.

In order to provide guidance for adjusting CT and SOT values, we offer some criteria for a good component. From the perspective of the users, there are several criteria for measuring the quality of components: reusability, changeability, cohesion, independency, coupling, and so on. Among these criteria, general design criteria, such as cohesion and coupling, are measured in our approach. Cohesion of a component is defined as the average dependency of the internal classes. Since super-classes and dedicated classes are included regardless of a component's main behavior, they are excluded in measuring cohesion. Coupling of a component is defined as the sum of dependencies among internal classes and external classes. By the cohesion and coupling information of an identified component, users can evaluate its quality.

VI. TOOL SUPPORT

For effectively performing the component identification process, it is necessary to provide an automated tool that systematically identifies domain components and connects existing CBD tools, such as a domain modeling tool or a component design & implementation tool, in order to support the full lifecycle of component development.

The component identifier was implemented by using JDK1.3. The component identifier is mainly composed of the Usage-Management Wizard and Algorithm-Performing Wizard, which are shown in Figs. 11 and 13. Through the Usage-Management Wizard, users specify weights of use cases and add or update object usages. After that, with the Algorithm-Performing Wizard, users perform an iterative identification process by adjusting SOT and CT values.

To support a full CBD process, the component identifier is integrated with the COBALT tool. Through O-O domain modeling with the COBALT tool, the component identifier generates domain components, which are inputted to the COBALT tool to proceed to further development phases. The COBALT tool is a CASE tool for supporting both the CD process and CBSD process. The tool is composed of the COBALT:Constructor [11], which provides a tool set for performing O-O domain modeling and developing and deploying EJB components, and the COBALT:Assembler [12], which provides a tool set for rapidly building a component-based software application by a visual plug&play assembly of EJB components.

VII. CONCLUSIONS AND FUTURE WORK

We have described a systematic component identification method and its procedure, which is one of the critical and difficult CBD processes, and provided the design and implementation of its support tool. With the help of the component identification tool, users can systematically identify reusable domain components. Since our component identification process is based on the O-O domain model that is widely used, users can easily apply the component identifier to existing O-O domain models without any additional effort. In addition, during domain modeling, users can concentrate on making O-O domain models without carrying out any preparations for the component identification process.

Due to integration of the component identifier and COBALT:Constructor, the tool efficiently supports the whole development cycle from the domain modeling process to the testing and deployment processes in an iterative and incremental manner. We believe that our tool has competitive power and it will be widely used for promoting the component industry in the future.

Until now, we have applied the identification algorithms to three practical examples for tuning or enhancing the algorithms. We found that tool users want to see object dependencies in a visual form. Additionally, due to the iterative component identification such as assigning weight values and applying algorithms, tool users want to intuitively know the changes of object dependencies when they assign new weight values. In a future work, we will develop a visualization mechanism and carry out a study on finding additional or supplementary domain information in order to more accurately reflect dependencies among objects.

REFERENCES

[1] D.F. D'Souza and A.C. Wills, *Objects, Components, and Frameworks with UML – the Catalysis Approach*, Addison-Wesley, 1999.

[2] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.

[3] M.S. Choi, Y.I. Yoon, and J.N. Park, "Study about a Component Identification Method Based on RUP," *Journal of Korea Information Processing Society*, Feb. 2002.

[4] TogetherSoft, *Features of Together*, http://www.togethersoft.com /products/controlcenter/ features.jsp, 2001.

[5] Computer Associates, *COOL:Joe 2.0 Product Descriptions*, http://www.cai.com/products/cool/joe /cooljoe_pd.pdf, 2001.

[6] Compuware, *About Uniface*, http://www.compuware.com/ products/uniface/about.htm, 2001.

[7] M.J. Kim, W.J. Lee, and G.S. Shin, "Development of COBALT (COmponent-Based AppLication deveLopment Tool) for Modeling and Constructing EJB Based Components," *IASTED-AI*2002, Feb. 2002.

[8] Object Management Group, *CORBA Components*, http://www.omg.org, Mar. 1999.

[9] Object Management Group, *Introduction to OMG's Unified Modeling Language* (*UML^{TM}*), http://www.omg.org/gettingstarted / what_is_uml.htm, 2002.

[10] Sun, *Designing Enterprise Applications with the Java^{TM} 2 Platform*, Enterprise Edition, Version 1.1, Mar. 2001.

[11] Y.J. Jeong, I.C. Yoon, M.J. Kim, W.J. Lee, S.J. Yoon, Y.J. Choi,

and G.S. Shin, "An Implementation of the Tool Supporting Design Pattern and Maintaining Alteration of Design Model for EJB Component Design," *Proc. of the* 2002 *Int'l Conf. on Software Engineering Research and Practice* (*SERP'*02), Lasvegas, USA, June 2002.

[12] Yoo-Hee Choi, Oh-Cheon Kwon, and Gyu-Sang Shin, "An Approach to Composition of EJB Components Using C2 Style," *Proc. of the EUROMICRO*, Sept. 2002.

[13] R. Monson-Haefel, *Enterprise JavaBeans*, Second edition, O'Reilly, 2000.

[14] P. Herzum and O. Sims, *Business Component Factoring: A Comprehensive Overview of Component-Based Development for Enterprise*, John Wiley & Sons, 2000.

[15] G. Caldiera and V.R. Basili, "Identifying and Qualifying Reusable Software Components," *IEEE Computer*, vol. 24, no. 2, Feb. 1991, pp. 61-70.

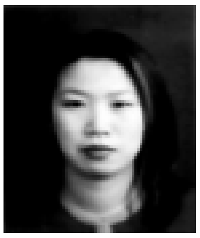
[16] Sanghyun Joo, Yongho Suh, Jaeho Shin, and Hisakazu kikuchi, "A New Robust Watermark Embedding into Wavelet DC Components," *ETRI J.*, vol. 24, no. 5, Oct. 2002, pp. 401-404.

[17] G. Caldiera and V.R. Basili, "Identifying and Qualifying Reusable Software Components," *Computer*, vol. 24, issue 2, Feb. 1991, pp. 61-70.

[18] Oh-Cheon Kwon, Seung-Yun Lee, and Gyu-Sang Shin, "A Product Line Approach through Architecture Reuse of a Component Assembly Process," *ICSE*2002 *Workshop*, USA, May 2002.

Woo-Jin Lee received his BS degree in Computer Science from KyungPook National University, Korea, in 1992, and the MS and PhD degrees in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Korea, in 1994 and 1999. He was a Senior Member of Engineering Staff in Electronics and Telecommunication Research Institute (ETRI) from 1999 to 2002. Since March 2002, he has been a faculty member of the Computer Science Department in Kyungpook National University, Daegu, Korea. His research interests include Requirements Engineering, CBD, and Modeling and Verification.

Oh-Cheon Kwon received the MS degree in software engineering from the University of Teesside, England, in 1994, and the PhD degree in computer science from the University of Durham, England, in 1998. He worked for SERI (Systems Engineering Research Institute)/KIST (Korea Institute of Science and Technology) from 1985 to 1997. He has been a Principal Researcher for ETRI since 1998. He was also a Visiting Researcher at IBM/RTP, North Carolina, USA, in 1991. He is currently involved in developing S/W architecture-based component technology and MDA-based S/W production technology. He has served as an Editor of Transactions of the Korea Information Processing Society, and an Assessor for qualifying the new S/W technology (KT Mark) sponsored by the Ministry of Science and Technology of Korea (MOST). His research interests include Component-Based Development (CBD), Model Driven Architecture (MDA), Web Services and S/W Reuse.



Min-Jung Kim received the MS degree in computer science from Sogang University, Seoul, Korea, in 2000. Since 2000, she has been a Member of Engineering Staff at Electronics and Telecommunications Research Institute, Daejeon, Korea. Her research interests include CBD (Component-Based Development), Web Services, and MDA (Model Driven Architecture).

Gyu-Sang Shin received the BS degree in statistics from Sung Kyun Kwan University, Korea, in 1981, and the MS degree in statistics from Seoul National University, Korea, in 1983, and the PhD degree in computer science from Chungnam National University, Korea, in 2001. He worked for Systems Engineering Research Institute (SERI), Korea as a Researcher between 1983 and 1995. Since 1996, he has been a Research Staff in ETRI, Korea. He is currently in charge of the Component Engineering Research Team. He has been engaged in the development of component-based development tools, real-time operating systems, video streaming servers, and object-oriented CASE tools. His research interests include Component-Based Software Engineering, Model Driven Software Development, CASE Tool and Multimedia Applications.