Component Identification Method with Coupling and Cohesion

Jong Kook Lee, Seung Jae Jung, Soo Dong Kim
Department of Computer Science
Soongsil University
Sangdo-dong, Dongjak-gu,Seoul, Korea
+82-02-820-0909
{jklee690/sjjung@selab, sdkim@comp}.ssu.ac.kr

Woo Hyun Jang, Dong Han Ham
Software Quality Evaluation Center
ETRI
Kajung-dong,Yusong-gu,Taejon,Korea
+82-42-860-1658
whjang/dhham@etri.re.kr

Abstract

Since the introduction of component-based development (CBD), effective component identification technique is known to be an important factor for successful CBD projects. As in CORBA Component Model by OMG, a component consists of one or more related objects, carrying out a homogeneous functionality. Most of the CBD methodologies utilize UML as the basic notational convention. Especially the component diagram or its variation is used to depict components. However, current CBD methodologies largely lack of systematic component identification algorithm that can be effectively used to group related use-cases and classes into components.

In this paper, we introduce component identification method that considers component coupling, cohesion, dependency, interface, granularity, and architecture. We also provide a case study on a large-scaled real CBD project, in which the proposed method was applied.

1. Introduction

Component software is the most promising method to solve software-crisis. Component-Based Software Engineering (CBSE) studies how to design component and defines what the process of developing component is [7, 8]. As CBSE is being evolved, some characters of component paradigm are emerged. Component must have well-defined interfaces. Component is so cohesive that it provides good service to users. The coupling of

components are so low that the system using components must have low complexity. The size of component is so adequate that system deployment and maintenance must be easy. Also, component is so very reusable that component must be used in many different systems. Therefore, component must be developed considering much aspect [1].

Many component development methodologies mainly support interface analysis and design. In CATALYSIS [2], to identify components, type model and action are used. CATALYSIS mainly focuses well-defined interface specification. Business Component Factory [3] uses the predefined component granularity. But it is not clear that the predefined component granularity support maintenance and reusability and low coupling, high cohesion.

In this paper, we suggest component identifying method considering coupling and cohesion and architecture and dependency, granularity. Developer may consider project and business-specific condition.

2. Criteria of Well-Defined Component

Well-defined component designs are driven by a variety of factors. In this paper, we emphasize that component granularity is the most important factor of design business component-based systems. Typically, the time and platform resources of inter-component communications are typically expensive. Thus, components must be larger rather than smaller. However, larger components by nature have more complex interfaces and possibility to be affected by change. If the component is large, the structure of the system is complex. Therefore a balance is important, depending upon the level of abstraction, likelihood of change, complexity of the component, and so forth [10]. Therefore, the principles of cohesion and coupling are important and essential factors for well-defined component design. In this study, we focus on high cohesion and low coupling to identify well-defined component.

3. Component Identifying Method

To identify well-defined component in chapter 2, we propose component identify method.

3.1. Overview

Component identification process is shown in Figure 1. At first, architecture design is performed to identify component. When we design architecture of system, we identify layer and subsystem. And then, we find use cases and classes for each subsystem. We can determine subsystem dependency using sequence diagram. Subsystem must be re-organized system to make subsystem dependency be less complex. Reorganized Subsystems can be performed by rearranging classes

0-7695-1083-3/01 $17.00 © 2001 IEEE

among subsystems. After subsystems are re-organized, our component identification method is performed.

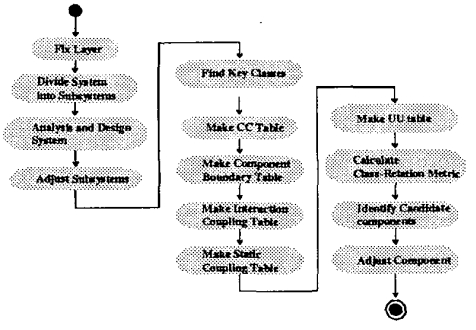


Figure 1. Component Identification Process

3.2. Defining Architecture

Architecture provides high-level view of system. Architecture serves as an integration mechanism for assembling software components [7]. And also, Architecture restricts the potential mismatches that might arise among component interfaces and thus offers useful constraints for assembling the components. Third, the possibilities for reuse are greatest at the architectural level where specifications are least constrained [8]. In this reason, to identify component, architecture must be considered at fist. To identify component, layering architecture is very important. Layering architecture has subsystem and subsystem dependency. Dividing system into subsystems decreases hard work to identify component because subsystem has smaller classes than whole system. Also well-defined architecture decreases component dependency. When component is identified in whole system, component dependency is very complex. If architecture is well defined, well defined subsystem dependency decrease component dependency.[Figure 2]

Component granularity is simply defined as fine-grained and coarse-grained. Coarse-grained component is our considering component. Fine-grained component is small and reusable component. In layered architecture, common subsystem define fine-grained component. Therefore, we may only consider coarse-grained component to identify component.

In system composed of well-defined component, component granularity level is defined. In Catalysis, iterative process defined system-specific component granularity level. In Business component Factory, the predefined granularity level is defined. Through architecture design, developer can well define component granularity level.

Architecture constrains to identify component. Protocol and middleware constrain component implementation. Performance decision in architecture design stage constraint component granularity. Performance and middleware decision can be used as added condition of identifying component.

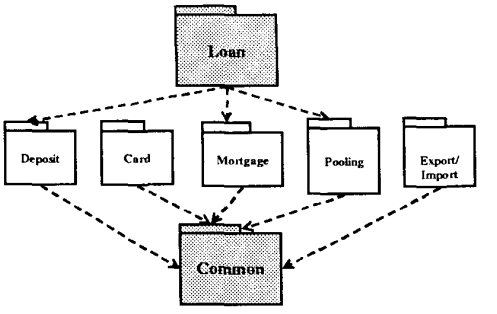


Figure 2. A Example of Subsystem in Banking System

3.3. Considering Component Cohesion

3.3.1. Finding Key Class

In UML Components, we find key classes to identify component at first and groups subsidiary classes with key classes. Finding key classes is important and essential task. But it is ambiguous what key classes are in UML Components. In Business Component Factory, component is defined as autonomous business concept. But its definition is also ambiguous. In this paper, we show process to find key classes systematically.

To identify key class in a use case, we can classify several functions into three function types in a use case. Function is performed as action unit by system within a use case. Function can be found from relation between actor and system in main flow of use case description or from sequence diagram. It can be method calls from system to actor or self method calls of system. Function type is classified into three types by function importance that can be presented as the degree of essential and core functionality in a use case. They are key function, subsidiary function and optional function. Key function is the core function in a use case. Subsidiary function is less important than key function and is performed with key function. Optional function is lowest important and is performed optionally with key function and subsidiary function. Function Type Table is used for classifying classes [Table 1].

Function Type	Function List
Key Function	
Subsidiary Function	
Optional Function	

Table 1. Function Type Table for a Use Case

Next, we classify classes in a use case into five class type considering feature of class participated in each function type, domain knowledge, and the responsibility of class (controller, entity and boundary class). To classify classes, sequence diagram or collaboration

diagram are helpful. If there is controller class which has heavy business logic or key entity class in a use case, its class type is A. Key entity class is best core and important class in a use case. If there is subsidiary entity class, its class type is B. Subsidiary entity class is less important than key entity class. But if controller class only delegate message to entity class and has light business logic, its class type is C. If there is boundary class performed in a use case, its class type is D. And if there are some classes that are the lowest key class in a use case, it's class type E [Table 2].

Class Type	Class Feature	Priority Weight
A	Key Entity Class Core Controller Class (it has heavy business logic)	5
B	Subsidiary Entity Class (it is performed with key entity class)	4
C	Controller Class (it has light business logic)	3
D	Boundary Class	2
E	The Lowest Key Class	1

Table 2. Class Type Table

A simple example about finding key class is shown following this. Class C3 has heavy business logic and Class C6 is key entity class in a use case. Therefore class type of C2 and C6 is A [Table 3].

Class Type	Class	Priority Weight
A	C3, C6	5
B	C4, C7	4
C	C5	3
D	C1, C2	2
E	C8, C9	1

Table 3. A Example of Class Type Mapping Table

3.4. Considering Coupling between components

When a message is passed between objects, the objects are said to be coupled. Classes are coupled when methods declared in one class use method or attributes of the other classes. Inheritance introduces significant tightly coupling between super class and their subclasses.

The number of message passing between two objects is commonly used as coupling [6]. We must distinguish "the number of method invoked" from the "number of method invocations". If the same method is invoked more than once and each invocation is counted separately, a distorted value for the measure can arise.

Therefore we must consider "the number of method invoked" as coupling between two classes. To calculate coupling value using "the number of method invoked", we consider sequence diagram.

If coupling value is defined as P_i, interaction coupling is defined as

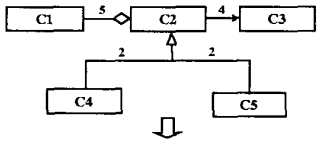
$$\text{Interaction Coupling} = \sum_{allUseCase} P_i$$

Class \ Class	C1	C2	C3	C4	C5
C1		3*1	2*1	1	1
C2			1	1	1
C3				3*1	2*2
C4					1
C5					

Table 4. Interaction Coupling Table

Next we consider static coupling. Static coupling is the coupling caused by class association, composition, inheritance. We assign to bidirectional association value 4, unidirectional association 3, aggregation 5, composition 6, parent-child relation 2, and sharing same parent class 1 [Figure 3].

An example of considering static coupling is shown in Figure 3.



Class \ Class	C1	C2	C3	C4	C5
C1		5	1	1	1
C2			4	2	2
C3				1	1
C4					1
C5					

Figure 3. An example of CC Table mapped with class diagram

Two classes coupling is defined as

Two classes coupling = interaction coupling * static coupling

3.5. Considering Component Interface

Component interfaces are the means by which components connect. Technically, an interface is a set of named operations that can be invoked by clients. Especially because component hides detailed information, clients only access through component interface into component. Component must provide single service to users. Developer determines what use case pair must be not included in a component and what use case pair must be included in a component. For example, some "include" use case has high reusability so that base use case must be separated from "include" use case to identify component. We provide Use Case-Use Case table. Developer fill out 'E' if two use cases are not included in a component, and fill out 'M' if two use cases are included in a component. A simple example of considering component interface is shown in Figure 4. and Figure 5.

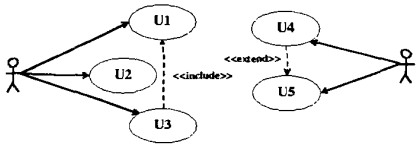


Figure 4. An Example of Use Case Diagram

Must be coupled : M, Exclusive : E

Use Case / Use Case	U1	U2	U3	U4	U5
U1		M	M	E	M
U2			M	E	E
U3				E	E
U4					
U5					

Figure 5. UU Table considering component interface

3.6. Considering Dependency between Components

Component dependency must be decreased. Runtime dependency is considered in previous chapter. We consider only import dependency. To decrease import dependency, after we identify component, component interfaces are managed in other package. We make the interface package [9]. Using interface package, many components cyclic dependency can be removed [Figure 6].

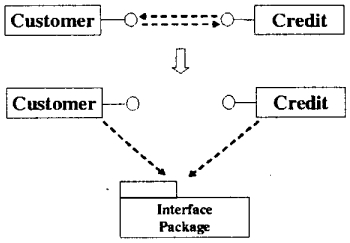


Figure 6. Dependency between Components

4. Component Clustering Algorithm

We provide component clustering algorithm to identify component using suggested component metrics.

4.1. Mathematical Basis

- **Clustering Relation**
  Let R be a binary relation on set A. R is an equivalence relation on A if R is reflexive, symmetric and transitive.
- **Cluster**
  If R is a clustering relation on a set A and a ⌝ A, then the cluster of a is defined to be:
  [a] = { b ⌝ A | (a,b) ⌝ R}
  In other words, [a] is the set of all elements which relate to a by R.
- **Theorem**
  Let A be a non-empty set and let R be a clustering relation on A. Then the distinct clusters of R partition A.

4.2. Class Relation Graph

- **Definition 1**
  node set N = { f_i | f_i is a class }

Next, we define the distance of nodes. We consider the semantic relation of class. In a simple model, if two classes are semantically related, relation is 0 and if two classes are semantically not related, relation is ∞. One-way relation (for example, calling and caller) is not defined. We define the distance of nodes as inverse of relation.

- **Definition 2**
  Dij = distance of classes pair
  We define edge set E
- **Definition 3**
  E = { Dij | Dij is a distance of classes }
  Then we define class relation graph G
- **Definition 4**
  class relation graph G = G(N,E)
  G represents the relation of classes. classes whose relation is very strong are represented as cluster in G. Cluster in G is the candidate of component. We want to extract cluster in G systematically. In Figure 7, the Number on line represents Dij

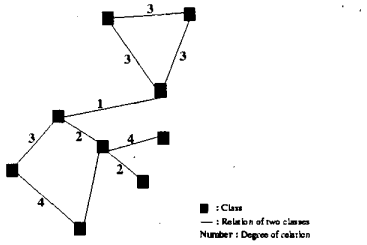


Figure 7. A Example of Graph G

To extract cluster, we choose some node n_j around node n_i. To identify component, we assume that the degree of relation of nodes in component is lager than d. If some node is related with several nodes, the node having maximum degree is chosen. In Figure 8, we assume that the degree of relation of nodes in component is larger that 20 so that i_n is related with n_j.

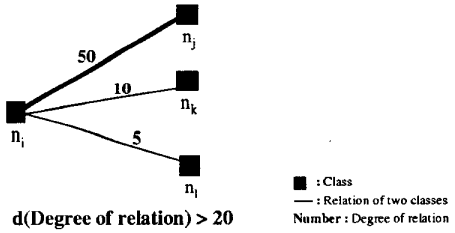


Figure 8. An Example of Related Node

If several nodes have the same degree of relation, we do not connect n_i with any other nodes and n_i is self-connected. And also if all the degree of relations with n_i are smaller than d, n_i is self-connected. Using our node-connection, every node in G have corresponding node. This correspondence is clustering function f_d over nodes in G. Using f_d, we can define the relation of any nodes n_i,n_j.

- **Definition 5**

Clustering Relation R of any nodes n_i,n_j is defined that there exists natural number n such that $f^n(n_i) = nj$ or $f^n(n_j) = n_i$.

Clustering Relation R means that nodes n_i,n_j in R is included in a cluster.

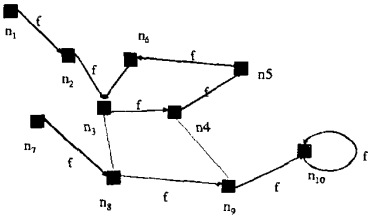


Figure 9. An example of node relation and clustering function

In Figure 9, $f^2(n_1) = n3$, $f^4(n_2) = n_6$. Therefore, n_1, n_2, n_3, n_4, n_5, n_6 make a cluster. To show that clustering Relation R implies a cluster, we prove that clustering relation is equivalence relation.

- **Proof**

If we prove that clustering relation is equivalence relation, we prove that clustering relation has reflexity, symmetricity and associativity.

Reflexity: n_iRn_i

$f^0(ni) = ni$. Then n_iRn_i

Symmetricity: $n_iRn_j \Rightarrow n_jRn_i$

For some n, $f^n(ni) = nj$ or $f^n(nj) = ni \Rightarrow f^n(nj) = ni$ or $f^n(ni) = nj$. Then $n_iRn_j \Rightarrow n_jRn_i$

Associativity: n_iRn_j and $n_jRn_k \Rightarrow n_iRn_k$

For some n, m, l, ($f^n(n_i) = n_j$ or $f^n(n_j) = n_i$) and ($f^m(n_j) = n_k$ or $f^m(n_k) = n_j$). If $f^n(n_i) = n_j$, Then ($f^m(f^n(n_j))$) $= n_k$ or $f^m(n_k) = f^n(n_i)$). $\Rightarrow f^{m+n}(n_i) = n_k$ or $f^{m-n}(n_k) = n_i$. If $f^n(n_j) = n_i$, Then ($f^{m-n}(ni)$) $= n_k$ or $f^m(n_k) = f^{-n}(n_i)$). $\Rightarrow$ $f^{m-n}(n_i) = n_k$ or $f^{m+n}(n_k) = n_i$. Then n_iRn_j and $n_jRn_k \Rightarrow$ n_iRn_k

Clustering relation is equivalence relation such that Graph G is divided with clusters. Clusters are candidate component. We can choose any d to select cluster in G. To choose d means that we choose clusters that are small or large. Therefore choosing d implies zoom-in and zoom-out effect to select cluster. In clusters in zoom-in and zoom-out effect, we can choose adequate cluster. To choose adequate cluster, we need the guideline to identify component and domain knowledge.

We can add other condition to Clustering relation. Domain expert can add domain-specific condition to clustering relation. Domain-specific condition must be equivalence relation. Domain-specific condition and Clustering relation are new relation to nodes. If Domain-specific condition is S, we can represent new relation as

$$R_{new} = S \cap R \text{ or } S \cup R$$

Clustering algorithm is represented as next pseudo code

```
D: the smallest distance of nodes in a component
N: the number of classes
Relation[N][N]: the degree of relation in classes
Cluster[]: cluster array

for i=0; i < n ;i++
next_node = i;
while( all nodes in a cluster are not found ){
// Find next node to next_node using node relation,D and other condition.
next_node = Next(next_node)
cluster[k++] = next_node
}

int Next(i)

temp = 0
  for j=0; j < n ; j++{
value = Relation[i][j]
 value1 = Relation[j][i]

 if( value > value1)
  if( value > temp )
   temp = value

 if( value1 > value)
  if( value1 > temp)
   temp = value1
}
return temp
```

Figure 10. Pseudo code of Clustering Algorithm

- **The metric of the relation of two classes**

The Degree of Relation of two classes is defined as:

(Class-Class Value based on cohesion)*(Interaction Coupling Value)*(Static Coupling Value)

4.3. The Guideline for Well-Defined Component

By using algorithm and metric proposed in chapter 5, we can identify candidate components. Now we provide guideline to identify well-defined components among candidate components.

- **Component encapsulates business process**
- **Stable component is placed into lower level than Unstable component**
- **The Dependency between components must be minimized**
- **The inner classes number of components is small enough to be managed**

5. Case Study and Assessment

Until now, we have discussed component identification method considering coupling, cohesion, dependency, interface, architecture and introduced component clustering algorithm with new metric to cluster classes. In this chapter, we present a case study of proposed component identification method and component clustering algorithm, and then discuss the practicality and efficiency of proposed techniques.

5.1. Defining Architecture

There are 9 subsystems in Banking System that we represent as a case study. Among them, we study Deposit Subsystem to identify components [Figure 11].

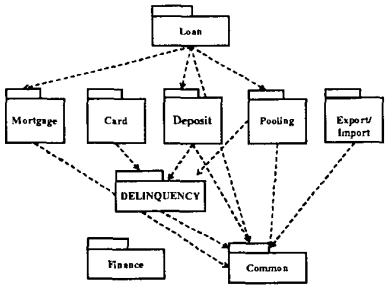


Figure 11. Banking System Architecture

5.2. Design Model for Deposit Subsystem

Figure 12 is the use case diagram of deposit subsystem.

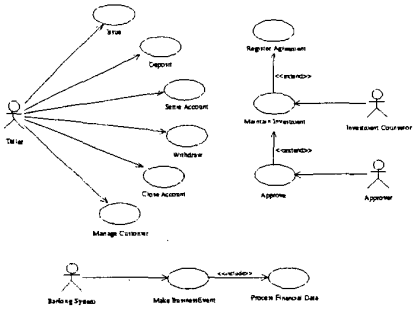


Figure 12. Use Case Diagram for Deposit Subsystem

Figure 13 is the class diagram of deposit subsystem.

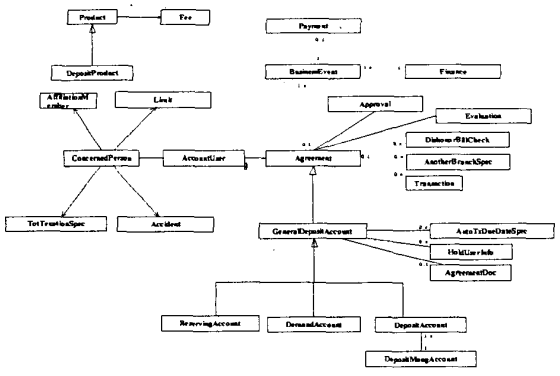


Figure 13. Class Diagram for Deposit Subsystem

When component identification method is performed, it is convenient for us to give ID for class name and use case name. And so we make ID table for Deposit Subsystem [Table 5].

ID	Class Name
C1	TotTexationSpec
C2	Transaction
C3	DemandAccount
C4	DepositAccount
C5	Customer
C6	Accident
C7	DepositProduct
C8	AnotherBranchSpec
C9	BusinessEvent
C10	Balance
C11	Payment
C12	DepositCounsel
C13	ProductCondition
C14	Fee
C15	Limit
C16	Approval
C17	Evaluation
C18	AddressAsignment
C19	Address
C20	AutoTxDueDateSpec
C21	PrizeWinning
C22	Finance
C23	Approval
C24	Evaluation

ID	Use Case Name
U1	Issue
U2	Deposit
U3	Settle Account
U4	Withdraw
U5	Close Account
U6	Maintain Investment
U7	Register Agreement
U8	Approve
U9	Make BusinessEvent
U10	Process Financial Data
U11	Manage Customer

Table 5. ID Table for Deposit Subsystem

5.3. Finding Key Class

To identify key class in a use case, we classify several functions into three function types in a use case. Function type is classified into three types (Key Function, Subsidiary Function, Optional Function) by function importance in use case. For example, Function Type Table for Issue use case is shown following this.[Table 6]

Function Type	Function List
Key Function	Create Accident Reporting, Check Accident Reporting
Subsidiary Function	Check Agreement Information, Search Concerned Person, Search Fee
Optional Function	

Table 6. Function Type Table for Issue use case

Next, we classify classes in a use case into five classes type considering feature of class participated in each function type and give priority weight for each class using Class Type Table [Table 2] and make Priority Weight Table for all use cases [Table 7]. Priority Weight Table for all use case is shown following this [Table 7].

C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19	C20	C21	C22	C23	C24
	4	5	4																
4	5	4																	
4	4	4														4		4	
4	4						5	4		4									
4	4						4			4									
4							4				5	5							
				5															
				5	4	4													
5													4	4					

Table 7. Priority Weight Table for All Use Case

5.4. Considering Component Cohesion

By using Priority Weight Table [Table 7], we can measure cohesion value between classes. Class-Class Table after calculating cohesion value is shown in Table 8.

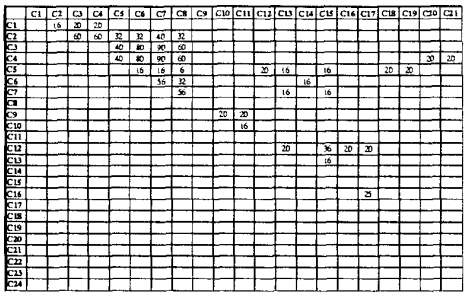


Table 8. CC Table after calculating cohesion value

5.5. Considering Coupling among Components

To determine interaction coupling value, we consider all sequence diagrams of deposit subsystem. In Table 9. (C3,C6) cell value represents the sum of message passing number between classes C3, C6 in deposit, get customer information and withdraw sequence diagram. In Table 10., value 4 represent that classes have bidirectional association and value 3 unidirectional association and value 2 inheritance.

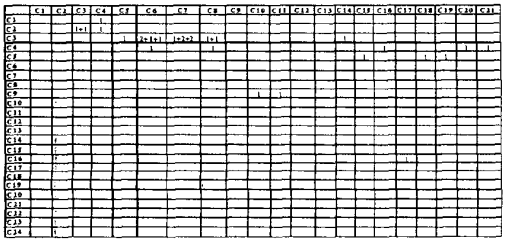


Table 9. Interaction Coupling Table

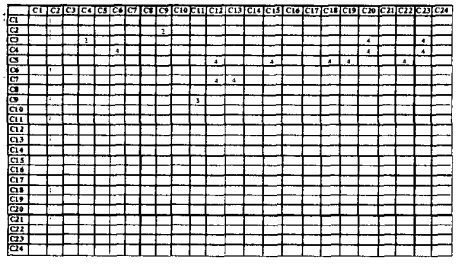


Table 10. Static Coupling

5.6. Considering Component Interface

We determine that Make BusinessEvent and Process FinancialData use case must be included in a component and Approve and Manage Customer use cases must not be included in a component.

Must be coupled : M, Exclusive : E

	U1	U2	U3	U4	U5	U6	U7	U8	U9	U10	U11
U1											
U2				M							
U3											
U4											
U5											
U6											
U7											
U8											E
U9										M	
U10											
U11											

Table 11. UU Table

5.7. Final Result: The degree of relation of classes

The cell value represents the degree of relation of two classes. The final result is show in Table 12.

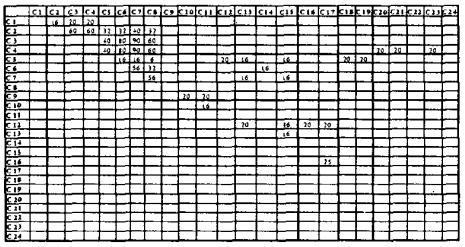


Table 12. Degree of relation table

5.8. Component Diagram for Deposit Subsystem

Using degree of relation table and clustering algorithm, we identify candidate components. Table 13 Represents candidate components. Candidate component1 represents customer management in Table 13. And candidate component2 is Deposit Management Candidate component3 is Account Management. Approval and Evaluation classes are not included in candidate components. The two classes are included in component by our guideline to identify adequate component. Approval and Evaluation classes must not be included candidate components because the service provided by approval and evaluation classes is not coincided with the service of candidate components. And the service provided by two classes has other processes with candidate components. Therefore the guideline that component encapsulates business process make two classes another components.

Candidate Component	Classes
Candidate Component1	Concernedperson, Limit, AccountUser TotalTexationSpec, Accident, AffiliationMember
Candidate Component2	Product, Fee, DepositProduct Agreement, DishonoreBillCheck, AnotherBranchSpec GeneralDepositAccount, AutoTxDueDateSpec HoldUserInfo, AgreementDoc, ReservingAccount, DemandAccount, Deposit Account, DepositAccount
Candidate Component3	BusinessEvent, Finance, Payment

Table 13. Candidate Components Table

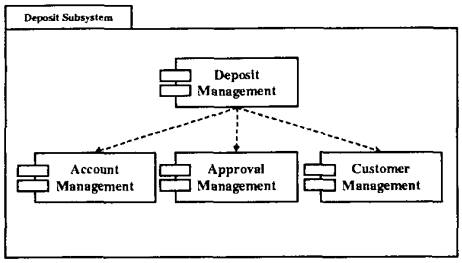


Figure 14. Identified Components in Deposit Subsystem

Identified components [Figure 14] show that our method is coincided with banking expert's experience. Deposit system executes deposit and withdrawal and financial processing is executed in deposit system. Identified components have deposit management component, account management component and customer management component. In aspect of reusability and adaptation, Separation of deposit and account components makes deposit subsystem evolve in new customer's needs and complex financial environment.

6. Conclusion

In this study, we proposed a component identification method that considers class cohesion, class interaction coupling, class static coupling. The method is flexible to reflect developer's domain knowledge by using class-class exclusion relation and use case-use case exclusion relation. Our clustering algorithm makes zoom-in and zoom-out effect and results in many clusters. To choose adequate components among many clusters, developer can use the domain knowledge and the guideline to identify component. Our clustering method is appropriate to determine component's granularity. Developer can add extra condition in the clustering method. In case study, we analyzed banking deposit system. The result of clustering shows that components identified by our method are coincided with domain knowledge and experience. We hope the proposed framework can be widely utilized in modeling components.

7. References

[1] Szyperski C., *Component Software: Beyound Object-Oriented Programming*, Addison Wesley Longman, Reading, Mass., 1998

[2] D'souza D. F. and Wills A. C., *Objects, Components, and Components withUML*, Addison-Wesley, 1998.

[3] Herzum P and Sims O, *Business Component Factory*, OMG, 2000

[4] Lee, Sang Duck, Yang, Young Jong, Cho, Eun Sook, Kim, Soo Dong, *"COMO : A UML-Based Component Development Methodology"*, Asia-Pacific Software Engineering Conference(APSEC99), Takamachu, Japan, PP. 54 – 61, Dec. 7-10, 1999.

[5] Software Quality Metrics for Object-oriented System Environments, USA, 1995

[6] Lionel C. Briand, John W. Daly, and Jurgen Wust, *"A Unified Framework for Coupling Measurement in Object-Oriented Systems"*, IEEE Transaction of Software Engineering, 1996

[7] C. Gacek, A. Abd-Allah, B.Clark, and B. Boehm, *"Focused Workshop on Software Architecture: Issue Paper"*, CSE Technical Report USC-CSE-94-499, Center for Software Engineering, Southern California University, April 1, 1994.

[8] Seongwoon Kim, *"I-Cube, An Architecture-Driven Component-Based Development Methodology"*, University of Illinois at Chicago, 2000.

[9] Philip Kruchten, *The Rational Unified Proces, Introduction*, Addison&Wesley, 1998

[10] Jon Hopkins, *"Component Primer"*, Communication of the ACM Vol. 43, No.10, October 2000.