A Reengineering Method from Procedural SW to Object-Oriented SW for SaaS in Cloud Computing

Moonkun Lee

The Division of Computer Science and Engineering

The Chonbuk National University

+82-63-270-3404

moonkun@jbnu.ac.kr

Abstract

Abstract:- One of the strong requirements for SaaS in cloud computing is the reengineering capability to transform procedural SW to object-oriented SW in case that there is no object-oriented SW available for a target application but procedural SW. Consequently it will be necessary to convert the procedural to the object-oriented automatically by quantifiably acceptable means of conversion that guarantees the quality of SaaS. This paper presents such a conversion method from C to C++ with the quantifiable means based on similarities for classes and inheritance required for SaaS. The first phase of the conversion is to simplify the C source code into a graph with a minimum number of tightly coupled components. The second phase is to generate all the possible groups of the class candidates from the graph. The third phase is to generate class signature similarities in each group to determine class inheritance. The last phase is to generate class and inheritance similarities between each group and the domain in the application. Compared with other approaches, this method gives SaaS experts with a comprehensive and integrated base to control selection of the best or optimal group of the class candidates for the application from cloud computing.

Keywords: *Cloud Computing, SaaS, SW Reengineering, Class/Inheritance Extraction, Similarity*

_____*****_____

1. Introduction

There are strong needs for reengineering procedural SW (PSW) to object-oriented SW (OOSW) in cloud computing, especially for SaaS [11, 12], since it is necessary to convert PSW to OOSW automatically when OOSW is not available is for the application but PSW. It implies that the objective of the reengineering prior to cloud computing is still valid in the time of cloud computing: modernizing legacy software for increase in software productivity, reduction in maintenance expenses, flexibility in adaptation to the new environment, etc [1]. Further the objectives have to enhance the new requirement that SaaS experts should be able to control the process of selecting the best or optimal OOSW for the application, based on some quantifiable means of measurements [11, 12].

Figure 1 shows the generic process for the reengineering as follows [1, 5]. It can be applied to the reengineering for SaaS:

1) Object extraction phase: In this phase, the building blocks or units of PSW are clustered into a number of meaningful sets, that is, object candidates, based on the degree of interconnectivities between global variables, used-defined variable types, functions and parameters in the PSW.
2) Class extraction phase: In this phase, classes are defined by extracting common features among related object candidates. The resulting classes have no class hierarchy among themselves. The inclusion relations among the classes based on the common features are not revealed yet.
3) Inheritance phase: In this phase, such inclusion relations among the classes are formalized to be *part-of* or *is-a* relations.

The formal relations are used to construct the hierarchical structure for inheritance among the classes, and the new common overlapped classes can be generated if necessary.

4) Persistence phases: In this phase, the characteristics and dynamic properties that are not in PSW are added in a new OOSW. The examples are the constructors and destructors of classes, their memory allocations and deallocations, etc.

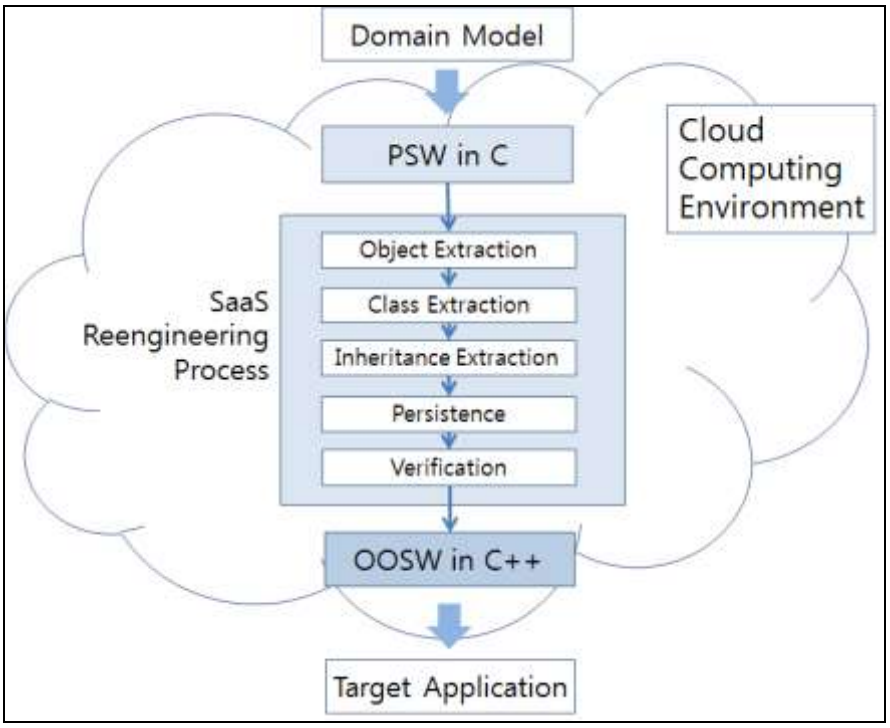

Figure SaaS Reengineering Process from PSW to OOSW in Cloud Computing

Prior to cloud computing, a number of researches were reported to transform PSW into OOSW mainly at the object and class extraction phases [2, 3, 5, 6, 7, 8, 10]. Further they only focused on producing only one deterministic pattern of target OOSW with respect to classes and inheritance. Consequently it was not possible for reengineers to consider other possible reengineering choices for the target OOSW. This is still the main issue for SaaS in cloud computing [11, 12]. Consequently it was not possible for SaaS engineers to control the process of selecting the best or optimal OOSW for the application based on some quantifiable means of measurements

In order to overcome the limitation, this paper presents a new and innovative conversion method that the engineer is able to select the best suitable OOSW for the application from groups of class candidates with a set of reengineering factors to determine the similarities between the candidates and the application domain for classes and inheritance. As shown in Figure 2, the method consists of the following phases:

1) Preprocessing phase: In preprocessing, PSW will be represented in a graph. Then the graph will be reduced by grouping its nodes with the tightly coupled relations, such as, variable-type, function call, memory-reference, etc. In this phase, the engineer can determine types of the grouping relations for the application.

2) Class extraction phase: In this phase, groups of all possible class candidates are generated in the first step. The number of groups appears to increase exponentially by the number of candidates, but does linearly due to the sequential dependency among the candidates. Once groups are determined, all the attributes (variables) and methods (functions) are

clustered together for the candidates and further abstracted. In this phase, the engineer can determine clustering criteria for attributes and methods.

3) Inheritance extraction phase: All the groups from the previous phase are organized in the abstraction levels of a special graph. At each level from the bottom, similarities among classes are determined to generate inheritance classes as supper classes. The results at the bottom will be reflected to the next top level from the bottom. It will repeat recursively up to the top of the graph. For the calculation of the similarity, the engineer can provide a set of signature similarity criteria in a quantifiable manner.

4) Decision Phase: With the given domain for the application, the similarities between the groups at each levels of the graph and the domain are measured with respect to the classes (vertical) and their inheritances (horizontal). The group with the best or optimal similarities will be selected as the best or optimal OOSW. Here the engineer can provide a set of vertical and horizontal similarity criteria in a quantifiable manner to select the best.

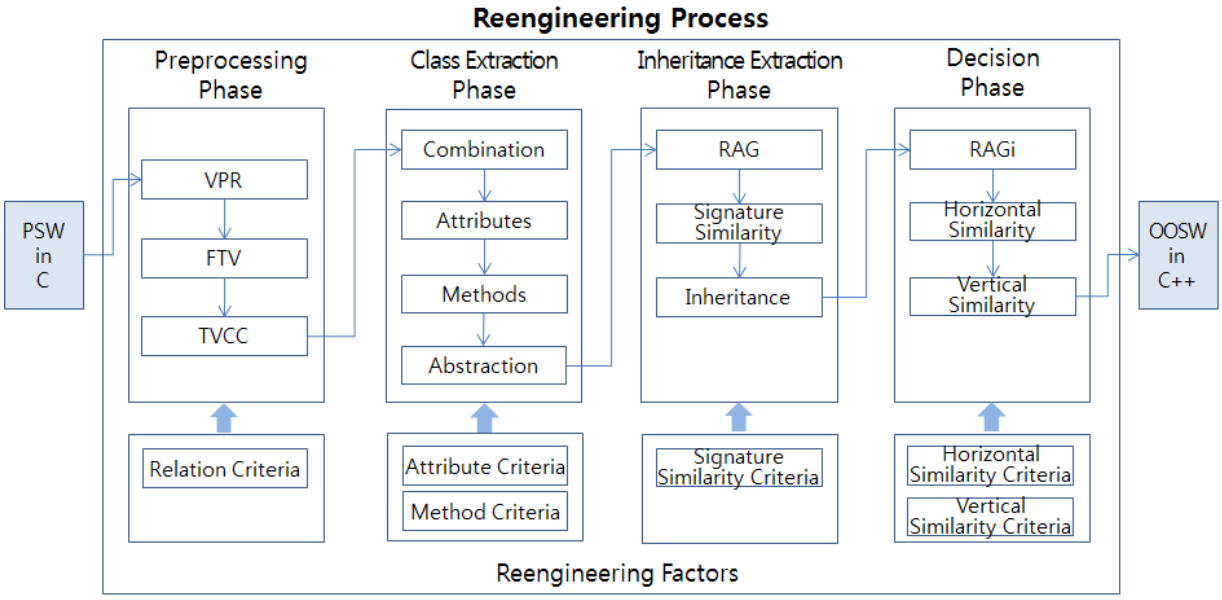

Figure Detailed Reengineering Process for Optimal Selection

As described, one of the main advantages of the method is to provide SaaS engineers with a comprehensive and integrated means to control the process of selecting the best or optimal OOSW for the application, based on different levels of reengineering factors in quantifiable manners.

Compared with other approaches [1, 2, 3, 4, 5, 8, 10, 11, 12], the method has the following innovative features:

1) Graphical model: It provides a graphical model for whole reengineering process, which makes the process clear and easy to comprehend.
2) Multiple candidates: It provides multiple groups of class candidates.
3) Minimum sets: It provides the minimum set of input graph in the preprocessing phase, of combinations in the class extraction phase, etc.
4) Reengineering factors: It provides four different types of reengineering factors at the phases of the processing, class extraction, inheritance, and the decision.
5) Decision criteria by SaaS engineer: The best or optimal OOSW is determined by the reengineering factors supplied by the

engineer. It means that the engineer has a strong influence on the selection.

The assumptions in the paper are as follows: there exists the documentation for the requirements of input PSW and a SaaS expert who can determine a domain model from the requirements. Extracting objects from PSW is not enough to define meaningful objects and classes. Therefore it is necessary to determine some domain model from the requirements and to determine similarities between the extracted classes and the classes in the model.

The paper is organized as follows. Section 2 presents related research. Section 3 defines graph models used in the paper. Sections 4, 5 and 6 describe each phase of extracting class candidates, extracting inheritance, and deciding the group of best or optimal class candidates. Section 7 shows the results of experiments and analysis. Section 8 discusses the conclusions and the future research.

2. Related Research

In COREM, PSW is transformed into OOSW after creating object modeling of PSW at abstraction level by referring information from both reverse-engineered knowledge and the domain-engineered modeling information by domain engineers [5].

Jin uses the relations among global variables, types and functions in order to extract objects and classes from PSW. A graph is constructed from the relations. The degree of a relation is represented by the frequency and the weight of each type of the relations. The clustering is performed with respect to the value of internal interconnectivity [7].

In the above approaches and others [2, 3, 6, 8, 10], the attributes of classes are commonly extracted by using function parameter, global variables and types. The methods of classes are extracted from functions or sliced portions of functions. Such a process is performed iteratively with respect to a group of object candidates. An expert is engaging in each step of the process. One of the problems in the approaches is that each different clustering sequence generates non-deterministically different results. Further only one group of object candidates is compared with a domain model.

In [11, 12], challenging issues and strategies for cloud computing are discussed by means of reengineering legacy applications. Especially it emphasizes requirements and criteria for business applications, but it does not describe technical issues of the reengineering, especially for conversion of PSW to OOSW. The requirements and criteria can be interpreted as a guideline for the reengineering performed by SaaS engineers. This paper will show one of the possible guidelines.

3. VPR and FTV Graphs

The graph model used in the paper is *Visual Program Representation* (VPR) [9], based on *Entity-Relation-Attribute* (ERA) model [4]. VPR is a graphical language used for visualization, analysis and transformation for a number of systems [13, 14].

3.1. VPR Graph

Each statement of SW in C and C++ is represented as a node in VPR graph. There are a number of node types for declaration and execution statements. Note that n_{name} and N_{name} imply the *name*d node or its type.

A relation between statements is represented as an edge with a weight. The weight implies the frequency of the relation. There are a number of edge types. Note that e_{name} and E_{name} implies the *name*d edge or its type.

G^{VPR} is a VPR graph with a set of nodes and a set of edges to represent input PSW.

3.2. FTV Graph

The *Function-Type-Variable* (FTV) graph G^{FTV} is a sub-graph of G^{VPR}, whose nodes are only of *function*, *data type*, and *global variable*, and whose edges are only of *function call*, *type reference* and global *variable reference*. G^{FTV} is to visualize and analyze

the information needed in extracting classes and inheritance from PSW.

Figure 3 shows FTV graph for a personnel management (PM) example written in C. There are three types of nodes: user-defined types, global variables, and functions. And there are three types of edges: type references (T), variable references (M), and function (C). The frequency of a relation is denoted at the edge after a colon.

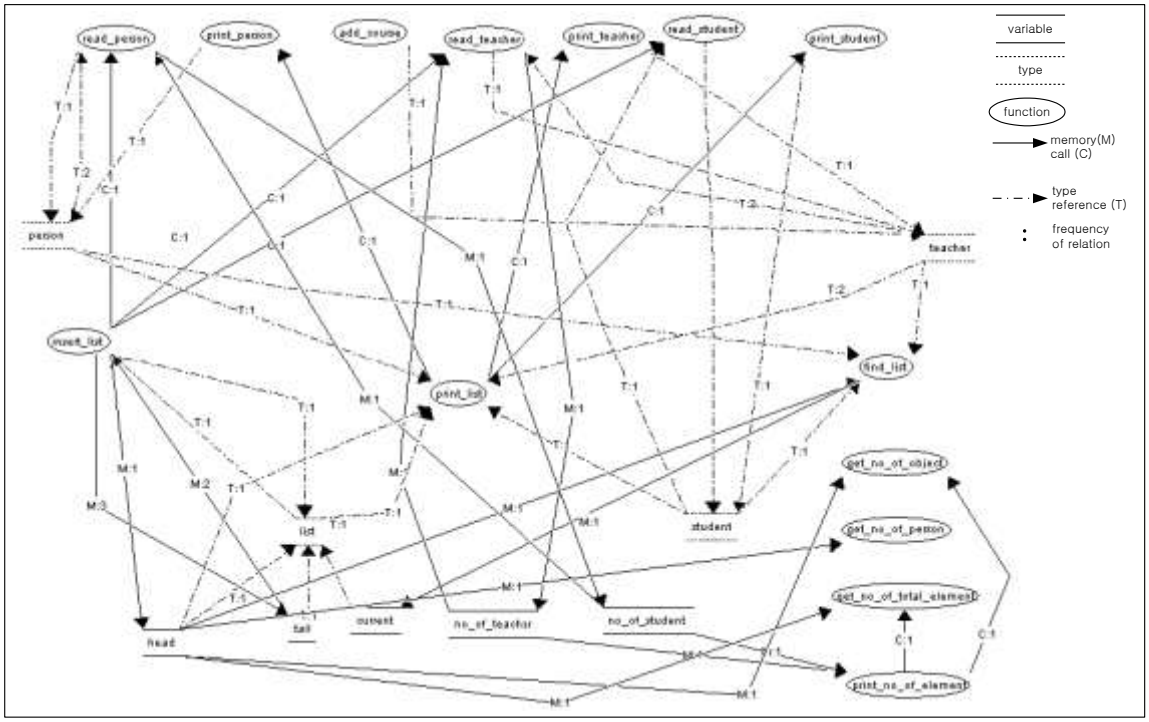

Figure 3 The FTV Graph for PM Example

3.3. TVCC Graph

FTV graph can be very complex. In order to reduce the complexity, a group of tightly coupled nodes can be considered as a composite node.

Firstly, the global variables of the same data type can be merged into a cluster. It means that the variables will fall in the same class and refer to the same functions. The approach can be similarly applied to the modified cases of the user-defined types. This type of clusters is defined as the *type-based variable connected component* (TVCC), denoting a group of nodes consisted of the global variables from G^{FTV} with the same type. The nodes and the graph of *TVCC* in G^{FTV} are represented as N_{TVCC} and G^{TVCC}, respectively. For example, the type *list* and the variables, *head*, *tail* and *current* of the *list* type in Figure 3 become the *list&head&tail&current* N_{TVCC} as shown the left corner of the bottom of Figure 4.

Secondly, the dynamic entities in the graph, that is, functions, can be pre-clustered. If a function is independently related to other node (i.e., there is only a single edge from the function to any other node), the function can be merged into the node as a cluster. This case is not found in Figure 3.

Figure 4 shows the TVCC graph from Figure 3. The number of nodes is reduced from 23 to 20: one cluster of a data type and three variables of the type.

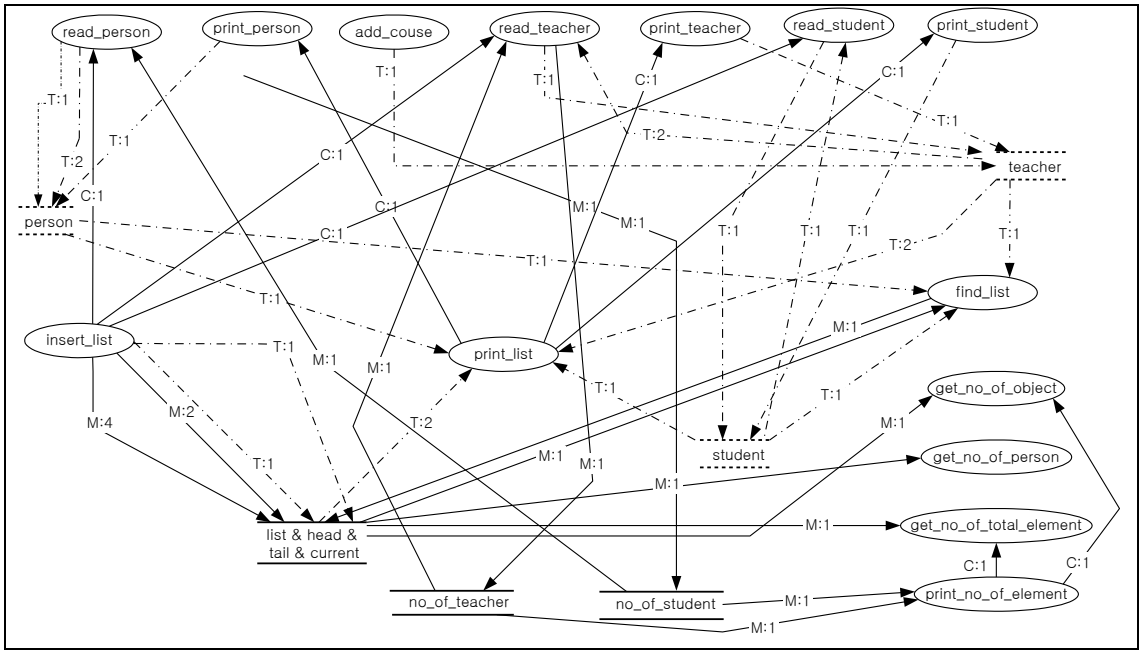

Figure 4 The TVCC Graph for PM Example

4. Class Extraction

This section describes the class extraction phase. The phase consists of four steps: *combination*, *attribute clustering*, *method clustering*, and *abstraction* described as follows.

4.1. Combination Step

The purpose of the step is to define the maximum number of groups for possible class candidates by generating all combinatorial cases for each type of clustering with respect to the number of TVCC nodes in G^{TVCC}. A graph G^{TVCC} from the previous section is input to the step, the number of N_{TVCC} in the graph is counted, and all the possible combinatorial cases to cluster N_{TVCC} are determined. The graph for each clusterering is denoted by $G_{a.b:(c_i,\ldots,c_i)}$, where *a* and *b* represent the identifier (ID) of the input graph and the number of clusters in the graph respectively, and $c_1,\ldots,c_i$ represent an ordered tuple of the number of TVCCs in the custers. The sum of $c_1,\ldots,c_i$ should be the number of TVCCs in the graph. For example, if the ID of a graph, the number of N_{TVCC} in the graph and the type of clustering combination are 1, 6, and three clusters with one N_{TVCC}, two N_{TVCC} and three N_{TVCC}, i.e., (1, 2, 3), respectively, the graph will be denoted by $G_{1.3:(1,2,3)}$. $G_{a.b}$ without the subscript (c_1, ..., c_i) implies either an arbitrary *cluster i* or a graph $G_{a.b}$ only with one cluster.

As the numbers of combinations for each type of clustering is obtained by the following formula:

$$\sum_{k=1}^{n}\sum_{j=1}^{y}\prod_{i=1}^{k-1}\binom{n}{m_i}$$

where *k* is the number of clustering types, *y* is the number of combinations in the clustering types, m_i is the number of TVCCs in the *i*-th cluster in the clustering type. Note that $\left|G_{a.b:(c_i,\ldots,c_i)}\right|$ for each clustering type represents the number of the combination cases for $G_{a.b:(c_i,\ldots,c_i)}$. The total number of combination for PM example is obtained as follows:

$$\left|G_{1.1:(6)}\right| + \left|G_{1.2:(1,5)}\right| + \left|G_{1.2:(2,4)}\right| + \left|G_{1.2:(3,3)}\right| + \left|G_{1.3:(1,1,4)}\right| + \left|G_{1.3:(1,2,3)}\right| + \left|G_{1.3:(2,2,2)}\right| + \left|G_{1.4:(1,1,1,3)}\right| + \left|G_{1.4:(1,1,2,2)}\right| + \left|G_{1.5:(1,1,1,1,1)}\right| + \left|G_{1.5:(1,1,1,1,1,1)}\right|$$

$$= \binom{6}{6} + \binom{6}{1} + \binom{6}{2} + \binom{6}{3} + \binom{6}{2} + \binom{6}{1}\binom{5}{2} + \binom{6}{2}\binom{4}{2} + \binom{6}{3} + \binom{6}{2}\binom{4}{2} + \binom{6}{4} + \binom{6}{6}$$

$$= 1 + 6 + 15 + 20 + 15 + 60 + 90 + 20 + 90 + 15 + 1 =$$

$$= 333$$

As shown above, the total number of combinations can be very large that the process of class extraction would be very complex task. However there exists at most one deterministic sequence of clustering for each clutering type as shown in the next step. Consequently at most one combination case can be generated for each clustering type. The maximum number of all combination cases is defined in the following formula:

$$f(x,y) = f(x,y-1) + f(x-y,y)$$

where,

$$\begin{cases} \forall y f(0,y) = 1 \\ \forall y f(x,y) = \forall x f(x,0) = 0 \quad (if \quad x < 0) \end{cases}$$

The initial values of x and y are the number of N_{TVCC} in G^{TVCC}. Compared with the previous formula, the total number becomes linear to the number of TVCCs in the graph.

4.2. Attribute Clustering Step

In the attribute clustering step, N_{TVCC}s in G^{TVCC} are clustered with respect to each combinatory case in sequence. The step can be considered as a process of extracting class candidates with static entities since the clustering is performed only on variables and types except functions. At the time of clustering N_{TVCC}, the following clustering criterion is required from a user or expert:

$$C^1 = < LIMIT, WET >$$

where,

$$WET = \{WET_{(N_{TVCC},N_{TVCC})}, WET_{(N_{TVCC},N_{spec})}, WET_{(N_{spec},N_{spec})}\}$$

where,

$WET_{(N_{TVCC},N_{TVCC})}$: *weight of* $e_{(N_{TVCC},N_{TVCC})}$,

$WET_{(N_{TVCC},N_{spec})}$: *weight of* $e_{(N_{TVCC},N_{spec})}$,

$WET_{(N_{spec},N_{spec})}$: *weight of* $e_{(N_{spec},N_{spec})}$, and

$$WET_{(N_{spec},N_{spec})} \le WET_{(N_{TVCC},N_{spec})} \le WET_{(N_{TVCC},N_{TVCC})}$$

Here C^1 and C^1_{LIMIT} imply the value of *LIMIT* and the minimum value of two N_{TVCC}s being clustered, respectively. Some unnecessary candidate groups can be discarded by selecting an appropriate value of C^1_{LIMIT}. The *weight of edge type* (WET) of C^1, that is, C^1_{WET}, implies how important each type of edges is. When two N_{TVCC}s are clustered, a different weight will be imposed on each type of edges between them. The values of C^1_{LIMIT} and C^1_{WET} should be ranged between 0 and 1.

The *value of relativity* (VR) to cluster two N_{TVCC}s is defined as follows:

$$VR_{(n_i,n_j)} = \prod_{l=i}^{j-1} \frac{w(e_{(n_l,n_{l+1})}) \times WET}{\sum(w(e_{(n_l,-)}) \times WET)}$$

where,

$$path(n_i, n_j) = < n_i, n_{i+1}, n_{i+2}, \cdots, n_{j-1}, n_j >$$
$$n_i, n_j \in N_{TVCC},$$
$$n_{i+1}, n_{i+2}, \cdots, n_{j-1} \notin N_{TVCC}$$

Here $e_{(n_k,-)}$ and $e_{(-,n_k)}$ mean all edges from and to an arbitrary node n_k.

When there are several paths and their *VR*s between node n_i and n_j, the *maximum VR* (MVR) represents the strongest interconnectivity between these nodes. *MVR* is similar to the most intensive path for the shortest distance between two nodes.

The algorithm for the attribute clustering is as shown in Algorithm 1. The inputs to the algorithm are a TVCC graph, the number of combinatorial cases from the combination step, and the clustering criterion. The time complexity of the algorithm is O(n^2).

```
Input  : G_i^TVCC , (a_y.1^x, a_y.2^x, ..., a_y.z^x) , C^1
Output : G_i.k^TVCC
While (ConditionOfCombNo( G_i^TVCC , a_y^x ))  {
  for each N_TVCC  { calculate MVR( N_TVCC , N_TVCC ) }
  if (max(MVR) > C^1_LIMIT )  {
    merge( N_fromMAX(MVR) , N_to:MAX(MVR) );
    restructure G_i^TVCC
  }
}
```

Algorithm The Attribute Clustering Algorithm

Here *ConditionOfCombNo*(G_i^{TVCC}, a_x^y) inspects if the given combinatorial case is applicable with the number of N_{TVCC}s in G_i^{TVCC}. The algorithm determines certain grouping cases where the clustering with the combinatorial case is not feasible. The first case is that the value of *MVR* is smaller than C^1_{LIMIT}. Since there is no strong relation between the N_{TVCC} , it is better not to generate a candidate group, which may produce some undesirable outcome, i.e., meaningless classes. The second case is that the already-clustered target N_{TVCC}s are not clusterable. Since the tightly coupled N_{TVCC}s are already clustered, partitioning the N_{TVCC}s reversely is not desirable.

Table 1 shows the output of the algorithm for PM example with the input (N_{TVCC}:N_{TVCC} =0.9, N_{spec}:N_{spec} =0.7, N_{TVCC}: N_{spec} =0.8). The table shows that there are only one clustering case for each clustering type, that is, $G_{1.1:(6)}$, $G_{1.2:(3,3)}$, $G_{1.3:(1,2,3)}$, $G_{1.4:(1,1,2,2)}$, $G_{1.5:(1,1,1,1,2)}$, and $G_{1.6:(1,1,1,1,1,1)}$. There are also non-applicable cases, that is, $G_{1.2:(1,5)}$, $G_{1.2:(2,4)}$, $G_{1.3:(1,1,4)}$, $G_{1.3:(2,2,2)}$, and $G_{1.4:(1,1,1,4)}$, since there are no clustering sequences for these clustering types in the algorithm. The total number of combination cases is now reduced almost in half.

Table 1 The Combinatorial Cases for PM Example

Combination	Class candidate groups
$G_{1.1:(6)}$	{person, teacher, student, list&head&tail&current, no_of_teacher, no_of_person}
$G_{1.2:(1,5)}$	Not applicable
$G_{1.2:(2,4)}$	Not applicable
$G_{1.2:(3,3)}$	{list&head&tail&current, person, no_of_person}, { no_of_teacher, teacher, student}
$G_{1.3:(1,1,4)}$	Not applicable
$G_{1.3:(1,2,3)}$	{list&head&tail&current, person, no_of_person}, { no_of_teacher, teacher}, {student}
$G_{1.3:(2,2,2)}$	Not applicable
$G_{1.4:(1,1,1,4)}$	Not applicable
$G_{1.4:(1,1,2,2)}$	{list&head&tail&current}, {person, no_of_person}, {student}, { no_of_teacher, teacher}
$G_{1.5:(1,1,1,1,2)}$	{list&head&tail&current}, {person, no_of_person}, {student}, { no_of_teacher}, {teacher}
$G_{1.6:(1,1,1,1,1,1)}$	{list&head&tail&current}, {person}, {no_of_person}, {student}, { no_of_teacher}, {teacher}

Figure 5 shows the group of two candidate classes for $G_{1.2:(3,3)}$: one with three N_{TVCC}s and the other with three N_{TVCC}s.

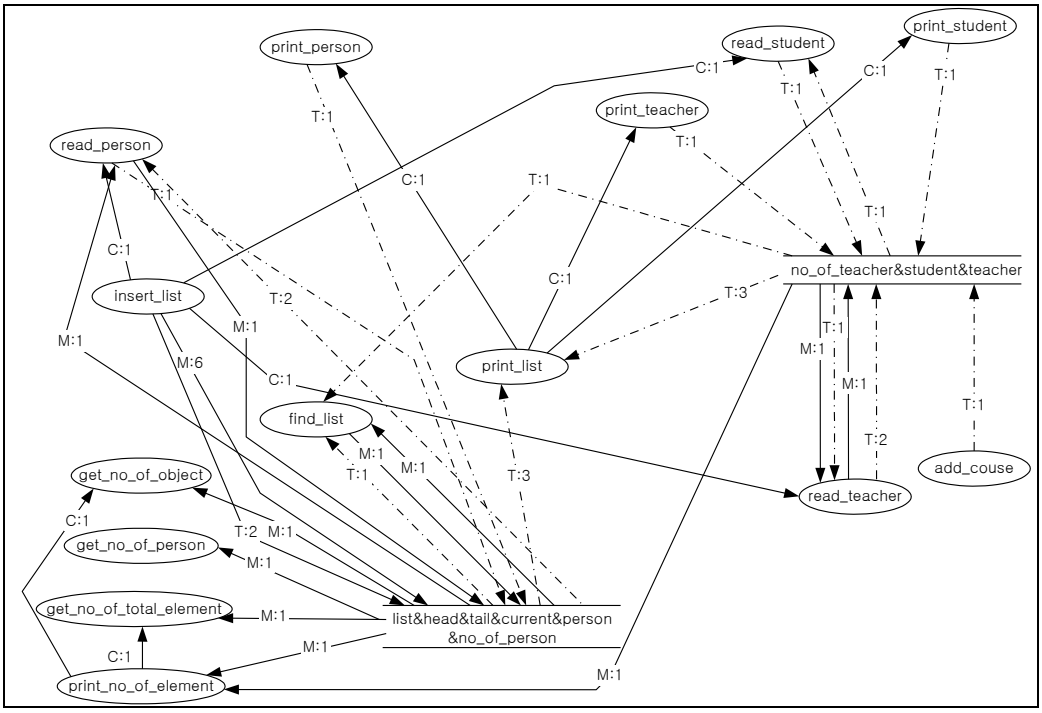

Figure 5 The Results of the Attribute Clustering for $G_{1.2(3,3)}$

4.3. Method Clustering Step

This step consists of two sub-steps: 1) cluster functions with a clustering algorithm and 2) handle the function remained after the algorithm by a domain expert.

4.3.1. Clustering with a Method Clustering Algorithm

As a result of the combination and the attribute clustering steps, the static elements of a graph G_i^{TVCC} , N_{TVCC}, are clustered. In the method clustering step, the dynamic elements, functions, are clustered into N_{TVCC}s. The functions are considered as methods in a class. The cluster of N_{TVCC} and N_{spec} is defined as *TVCC-extended-to-function* (TEF) node and represented as N_{TEF}. The graph with

N_{TEF} is called G^{TEF}.

For function clustering, the following clustering criterion is input from a user:

$$C^2 =< LIMIT, EWET >$$
where,
$$EWET = \{EWET_{<N_{TEF}, N_{spec}, FORWARD>}, EWET_{<N_{TEF}, N_{spec}, BACKWARD>}, EWET_{<N_{spec}, N_{spec}, FORWARD>}, EWET_{<N_{spec}, N_{spec}, BACKWARD>}\}$$

The *LIMIT* of C^2, C^2_{LIMIT}, is the minimum value for N_{spec} to be clustered. The *extended WET* (EWET) of C^2 is the weight of type and the direction of the edge. The values of C^2_{LIMIT} and C^2_{EWET} are ranged between 0 and 1.

A domain expert can adjust the degree of cohesion among nodes with C^2_{LIMIT}. If the real small value of C^2_{LIMIT} is selected, all functions can be clustered to N_{TEF}. In this case, there exists weak cohesion between loosely coupled N_{TEF} and clustered functions. Conversely, if the very large value of C^2_{LIMIT} is selected, all functions can not be clustered to N_{TEF}. In this case, there exists strong cohesion between tightly coupled N_{TEF} and clustered functions.

The remaining functions, which are not clustered to any N_{TEF} due to the high C^2_{LIMIT} value, are called *procedure remainder* [7]. It will be discussed in Section 4.3.2.

In comparison, a *writing* relation (or a *called* relation) has a higher weight than a *reading* relation(a *calling* relation). In addition, the weight of the relation between N_{spec} and N_{TEF} is higher than that of the relation between N_{spec} and N_{spec}. Such differences are due to whether a relation between nodes is direct or indirect. Such a type of a relation may imply the type function members to be accessed, i.e., *private* or *public* in the C++ notion.

Using a clustering criterion C^2, the VR between N_{TEF} and N_{spec} is obtainable. Since it has the value of the direction of an edge, it is defined as the *extended VR* (EVR) as follows:

$$EVR_{(n_i, n_j)} = \prod_{l=i}^{j-1} \frac{w(e_{(n_l, n_{l+1})}) \times EWET}{\sum(w(e_{(n_l, -)}) \times EWET)}$$
where,
$$path(n_i, n_j) =< n_i, n_{i+1}, n_{i+2}, \cdots, n_{j-1}, n_j >$$
$$n_i \in N_{spec}, \quad n_j \in N_{TEF}$$

When there exist several paths and EVR values from node n_i to n_j, the *maximum EVR* (MEVR) among the node represents the EVR between two nodes.

The degree of relativity to determine probabilistically of which N_{TEF} each N_{spec} should belong to is defined as follows:

$$DR(n_i, n_x) = \frac{MEVR(n_i, n_x)}{\sum MEVR(-, n_x)}$$
where,
$$0 \le DR(n_i, n_x) \le 1$$
$$n_x \in N_{spec} : \text{source node}$$
$$n_i \in N_{TVCC} \cup N_{TEF} : \text{target node}$$

Here MEVR(-, n_x) implies all N_{TEF} or N_{TVCC} which have some relations to the node n_x.

The method clustering algorithm requires a group of class candidates from the clustering step, $G_{i,k}^{TEF}$, and a clustering criteria, C_2. It processes each function determinately in any order as shown in Algorithm 2.

```
Input  : G_{i.k}^{TEF} , C^2
Output : G'_{i.k}^{TEF}  (graph clutsered)
EndFlag := false
while ( ! EndFlag ) {
   for each  N_spec  { calculate EVR( N_TEF , N_spec ) }
   MEVR := max(EVR)
   N_from , N_to      := getNodeOfMEVR()
   DR := getDR( N_from , N_to )
   if ( DR >  C^2_LIMIT  )  {
       merge( N_from , N_to )
       restructure  G_{i.k}^{TEF}
   }
   else { EndFlag := true }
}
```

Algorithm The Method Clustering Algorithm

Here *getNodeOfMEVR*() obtains the N_{TEF} and N_{spec} with MEVR. If the DR of the N_{TEF} and N_{spec} is higher than C^2_{LIMIT}, the two nodes are merged and the algorithm is applied iteratively to the updated graph with the merged nodes. The time complexity of algorithm is $O(n^2)$.

The output of the algorithm for PM example with an input (*LIMIT*=0.0000001, $EWET_{<N_{spec},N_{spec},FORWARD>}$ = 0.6, $EWET_{<N_{TEF},N_{spec},FORWARD>}$ = 0.8, $EWET_{<N_{spec},N_{spec},BACKWARD>}$ = 0.8, $EWET_{<N_{TEF},N_{spec},BACKWARD>}$ = 1.0) is shown in Figure 6.

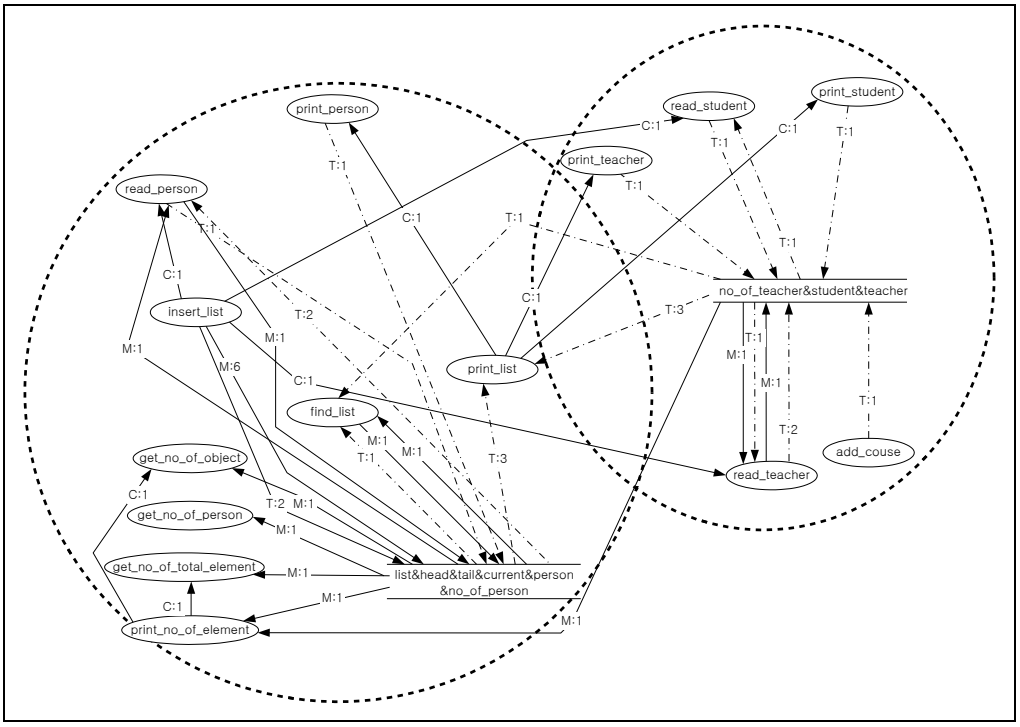

Figure 6 The Result of the Method Clustering for $G_{1.2(3,3)}$

4.3.2. Clustering by a Domain Expert

In the paper, the remaining functions from the method clustering step are handled by a domain expert in two ways: 1) the expert decides which class candidate, N_{TEF}, each function is to be clustered into, and 2) the expert determines the functions that do not

belong to any candidate at all. If a function is decided to belong to a certain candidate, it follows the same procedures applied to other class candidates. If a function is decided not to belong to any candidate, the function can become an individual function member of a class without any attribute.

In PM example, all functions are clustered into all N_{TEF} because a small value of C^2_{LIMIT} is given. If a large value is given, there will be some remaining functions. A domain expert can control the degree of cohesion between functions and N_{TEF} with C^2_{LIMIT}.

4.4. Abstraction step

Among class candidates among $G_{a.b:(c_1,\ldots,c_i)}$ s, there can be abstraction relations, resulting a *reverse acyclic graph* (RAG), G^{RA}. The nodes of G^{RA} graph have the generalization (abstraction) and specialization (actualization) relations among class candidates in each group with respect to only static element, N_{TVCC}, as shown in Figure 7. The node in the higher level is the generalization of the node(s) in the lower level. Likewise the node(s) in the lower level is the actualization of a node in the higher level. Due to its size, the methods are shown in the bottom layer only. These methods are included in the classes at other layers in the direction of edges.

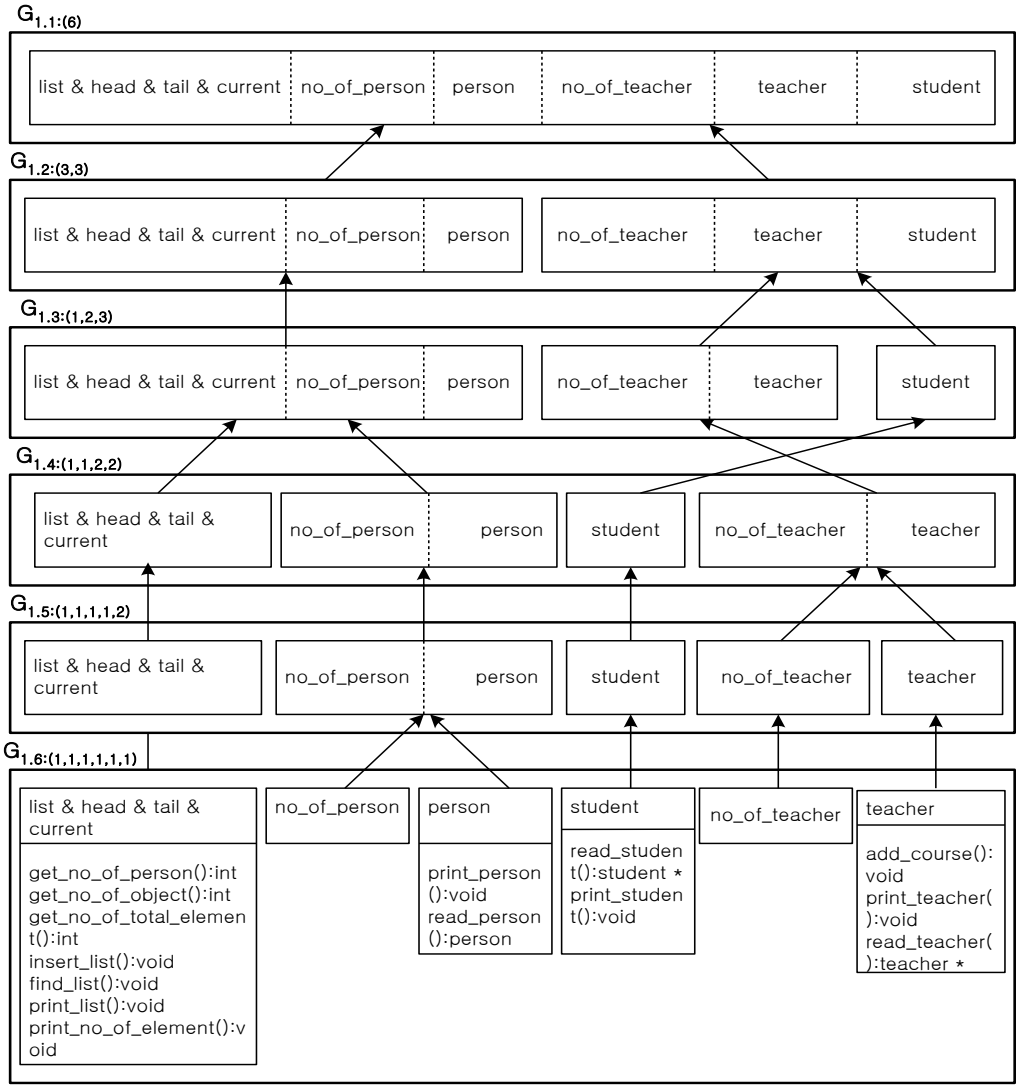

Figure 7 A RAG Graph for PM Example

5. Class Inheritance Extraction

The purpose of the inheritance extraction phase is to extract class hierarchy among class candidates from Section 4. The phase is based on an idea of class similarity among classes in each group of class candidates. If there is a high degree of similarity between two classes, there is a possibility of a supper class being generalized from them. This concept is applicable recursively to new super classes.

5.1. Similarity

The *degree of signature similarity* (DSS) is the measurement of similarity between two classes. DSS is extended from the *overall similarity factor* (OSF) [7]. The OSF is applicable only to measure the similarity between one object and a class, but DSS can be applied to measure the similarities between *n* classes and *m* classes. The definition of DSS is as follows, where the factors of OSF are redefined for this application:

$$DSS = sf_c * \left(\begin{array}{l} w_{an} * \left(\frac{\text{Sum of maximum } sf_{an} \text{ of each attribute}}{\text{Number of attributes}} \right) + \\ w_{at} * \left(\frac{\text{Sum of maximum } sf_{at} \text{ of each attribute}}{\text{Number of attributes}} \right) + \\ w_{mn} * \left(\frac{\text{Sum of maximum } sf_{mn} \text{ of each method}}{\text{Number of methods}} \right) + \\ w_{mt} * \left(\frac{\text{Sum of maximum } sf_{mt} \text{ of each method}}{\text{Number of methods}} \right) + \\ w_{mpt} * \left(\frac{\text{Sum of maximum } sf_{mpt} \text{ of each method}}{\text{Number of methods}} \right) + \end{array} \right)$$

Here sf_c is the similarity factor (SF) between class names based on some special semantic rules. *An*, *at*, *mn*, *mt*, and *mpt* imply attribute name, attribute type, method name, method return type, and method parameter type, respectively. Sf_{an}, sf_{at}, sf_{mn}, sf_{mt}, and sf_{mpt} are similarities for respective factors. And w_{an}, w_{at}, w_{mn}, w_{mt}, and w_{mpt} are the weight values for those factors, respectively.

The values of SFs in class signature are decided by a domain expert with respect to the syntactic and semantic interpretation on the element of classes. To obtain DSS, the following four conditions should be satisfied: 1) the sum of all weight values is 1, 2) the sum of sf_c between classes is $0 \le sf_c \le 1$, 3) the similarity between types is based on their names, and 4) the similarity between parameters is based on the methods they belong to.

5.2. Inheritance

In order to find inheritance relations among class candidates in each group of class candidates from Section 4, DSS is applied to them, resulting that a $n \times n$ matrix S holds the values of DSS between classes i and j, where n is the number of the class candidates, and i and j are the *id*s of the class candidates. The matrix is symmetric and the values in diagonal are 1.

The similarity for $G_{1.2:(3,3)}$ in the example is shown in Table 2. In order to obtain DSS, the weights of attributes and methods are defined uniformly to be (W_{an}=0.2, W_{at}=0.2, W_{mn}=0.2, W_{mt}=0.2, W_{mpt}=0.2) for demonstration purpose. The value of similarity indicates there is a possibility of a super class generalizable from two class candidates. The result of actualization of this similarity is shown as a super class in the second layer of Figure 8.

Table 2 A DSS Matrix for Class Candidates in $G_{1.2:(3,3)}$ of PM example

	{list&head&tail&current} {no_of_person}{person}	{no_of_teacher}{teacher} {student}
{list&head&tail&current} {no_of_person}{person}	1	0.47
{no_of_teacher}{teacher} {student}	0.47	1

5.3. Reverse Acyclic Graph with Inheritance

Similar to RAG in the attribute clustering step, a *reverse acyclic graph* for groups of class candidates with inheritance (RAGi) can be generated. Figure 8 shows the RAGi graph G^{RAi} for PM example. In the figure, a super class, unnamed so far, with the *person* attribute is generalized from the bottom layer to the second layer from top. Sub-classes are actualized at each layer respectively.

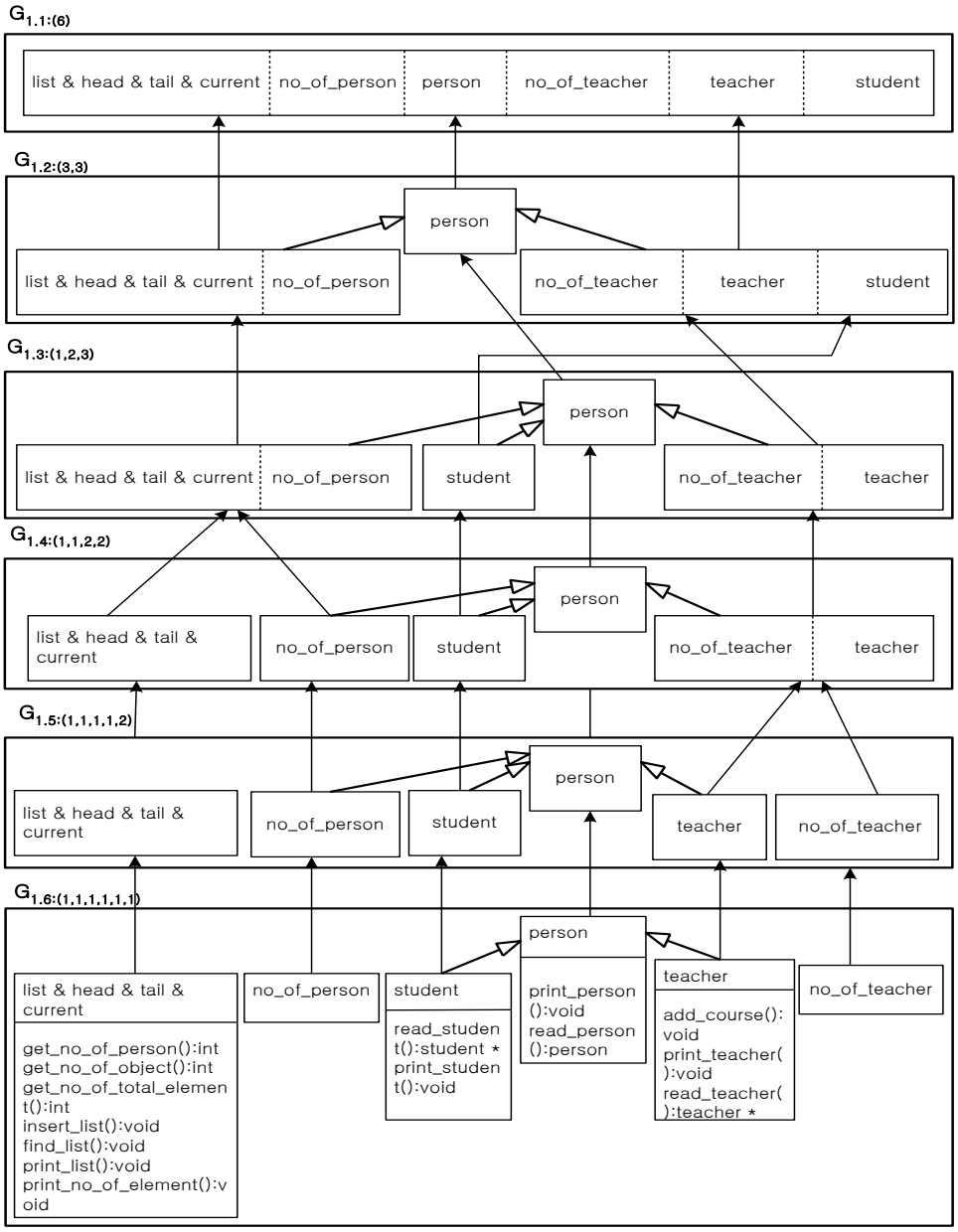

Figure 8 A RAGi Graph for PM example

6. Decision Step

The purpose of the decision phase is to assist a reengineer to determine a group of the best or optimal class candidates from section 5 by measuring the similarities between each group of the class candidates and the classes in a domain model constructed by a domain expert.

6.1 Two-Dimensional Similarity: Class Group Similarity

Two-dimensional similarity is composed of the following horizontal and vertical similarities.

6.1.1 Horizontal Similarity

The horizontal similarity is the similarity between the *i*-th group of class candidates from RAG-GCCI and the classes of domain model, defined as follows:

$$S^{CCG} = \sum_{k=1}^{m}\sum_{l=1}^{n}\left(S_{C_{(k,l)}}^{CCG_i} + S_{I_{(k,l)}}^{CCG_i}\right).$$

Here *k* and *l* represent the number of the candidates and the number of the classes, respectively. $S_C^{CCG_i}$ is a matrix implying *class similarities* between the *m* class candidates in the group and the *n* classes in the model with respect to attributes and methods. It is obtained by measuring the degrees of DSS of the *m* candidates to the *n* classes. $S_{C_{(m,n)}}^{CCG_i}$ represents class similarity between the *k*-th candidate and the *l*-th class in $S_C^{CCG_i}$.

$S_I^{CCG_i}$ is a matrix implying *inheritance similarities* between the *m* candidates and the *n* classes, defined as follows:

$$S_I^{CCG_i} = S_C'^{CCG_i} \times \left(\frac{Min\left(S_{CI}^{CCG_i}, S_{CI}^{HM}\right)}{Max\left(S_{CI}^{CCG_i}, S_{CI}^{HM}\right)}\right)$$

where $S_C'^{CCG_i}$ represents the class similarity only for the classes with inheritance from $S_C^{CCG_i}$, and $S_{CI}^{CCG_i}$ and S_{CI}^{HM} imply the ratios of inheritance for the candidates and the classes, respectively, defined as follows:

$$S_{CI}^{CCG_i} = \frac{CCG_{CI}}{CCG_C + CCG_{CI}}$$

$$S_{CI}^{HM} = \frac{DM_{CI}}{DM_C + DM_{CI}}$$

Here CCG_{CI}, CCG_C, DM_{CI}, and DM_C are obtained by adding all inheriting class members (*CI*), that is, attributes and methods in the group, and all inherited class members (*C*) for each the group (*CCG*) and the model (*DM*), respectively. $Min\left(S_{CI}^{CCG_i}, S_{CI}^{HM}\right)$ and $Max\left(S_{CI}^{CCG_i}, S_{CI}^{HM}\right)$ are the minimum and maximum values between $S_{CI}^{CCG_i}$ and S_{CI}^{HM}. $S_{CI}^{CCG_i}$ represents the degree of commonality among classes with inheritance in the group. S_{CI}^{HM} represents the degree of commonality

among classes with inheritance in the model. $S_{I_{(k,l)}}^{CCG_i}$ represents inheritance similarity between the *k*-th candidate and the *l*-th class in $S_I^{CCG_i}$.

Tables 3 shows $S_C^{CCG_i}$ for PM example with respect to a domain model for the example shown in Figure 9. The domain model is designed to demonstrate the approach in the paper. $S_C^{CCG_i}$, $S_I^{CCG_i}$, $S_{CI}^{CCG_i}$ and S_{CI}^{HM} are discarded due to their size.

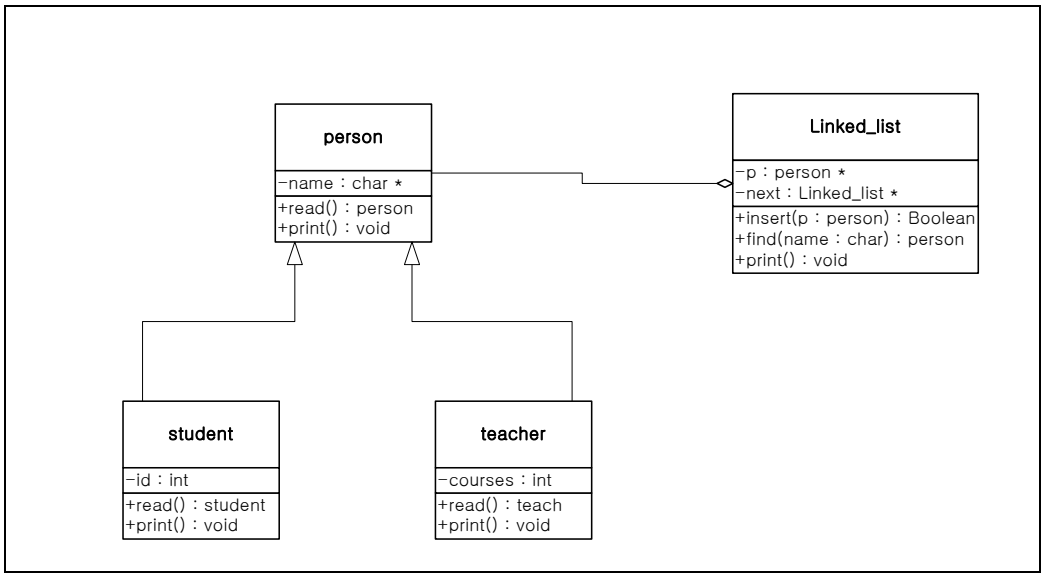

Figure 9 A Domain Model for PM Example

Table 3 S^{CCG_i} for PM Example

G	S^{CCG}
$G_{1.1:(6)}$	0
$G_{1.2:(3,3)}$	2.06
$G_{1.3:(1,2,3)}$	2.99
$G_{1.4:(1,1,2,2)}$	2.95
$G_{1.5:(1,1,1,1,2)}$	2.61
$G_{1.6:(1,1,1,1,1,1)}$	2.93

6.1.2 Vertical similarity: class hierarchy similarity

The vertical similarity is the similarity between the subset (C_g) of class candidates in the *i*-th group of class candidates from the previous step and each set (C_m) of classes in the same class hierarchy from the domain model. There will be a best or optimal C_g among the groups from RAG-GCCI for each C_m, where C_g is most similar with C_m in inheritance. This similarity is defined as follows:

$$S_{CIG}^{CCG_i} = \sum_{k=1}^{m}\sum_{l=1}^{n}\left(S_{C_{(k,l)}}^{''CCG_i} + S_{I_{(k,l)}}^{''CCG_i}\right)$$

Here k and l represent the number of the candidates and the number of the classes respectively. And $S_C^{''CCG_i}$ and $S_I^{''CCG_i}$ represent the degrees of class similarity and inheritance relation between C_g and C_m, respectively. These values are obtained from $S_C^{CCG_i}$ and $S_I^{CCG_i}$ only for those C_g and C_m. $S_{C_{(k,l)}}^{''CCG_i}$ and $S_{I_{(k,l)}}^{''CCG_i}$ represents class and inheritance similarity between the k-th candidate and the l-th class in $S_C^{''CCG_i}$ and $S_I^{''CCG_i}$, respectively.

Table 4 shows $S_{CIG}^{CCG_i}$ for PM example. It shows that $G_{1.4:(1,1,2,2)}$ has the highest value with the domain model. Note that there is only one set of classes with inheritance in the domain model.

Table 4 $S_{CIG}^{CCG_i}$ for PM Example

G	S^{CCG}: {person, student, teacher}
$G_{1.1:(6)}$	0
$G_{1.2:(3,3)}$	1.62
$G_{1.3:(1,2,3)}$	2.47
$G_{1.4:(1,1,2,2)}$	2.81
$G_{1.5:(1,1,1,1,2)}$	2.45
$G_{1.6:(1,1,1,1,1,1)}$	2.78

6.2 Decision

The decision for the group with the best or optimal class candidates is made by comparing both the horizontal and vertical similarities. The horizontal similarity represents the overall similarity between a group of class candidates and the classes in the domain model. The vertical similarity represents the class hierarchy similarity for each class hierarchy between the candidate group and the domain. As shown in Table 3, it cannot be guaranteed that $G_{1.3:(1,2,3)}$ with the highest value for horizontal similarity is selected as the group of best class candidates, since its value for the vertical similarity in Table 4 is relatively less than that of $G_{1.4:(1,1,2,2)}$. However there is a strong possibility that $G_{1.4:(1,1,2,2)}$ with the highest value of vertical similarity in Table 4 can be selected as the group with optimal class candidates, since its value for horizontal similarity in Table 3 is not much less than that of $G_{1.3:(1,2,3)}$. Consequently, if there are a number of class hierarchies in the domain model, there is a possibility of selecting a subset of partially best or optimal class candidates from each group of class candidates in order to construct a complete group with the overall best or optimal class candidates for the domain model.

7. Experiments and analysis

A number of experiments have been performed to demonstrate the feasibility and efficiency of the approach in the paper. Table 5 describes some randomly selected input PSWs. The data in the tabel reveals the following four facts: 1) there is a large variation for the number of clusters in the preprocessing and class extraction steps with respect to the size of the PSW, 2) the size of the PSW is not a main factor to determine the number of combinatorial groups of class candidates, which is represented relatively by the value in the method clustering entry of the table, 3) the main determining factor for the number is the number of TVCC clusters, and 4) there is a strong dependency between the number of global data types and the number of new super classes. The data shows that the numbers of clusters and classes are relatively small and manageable with respect to the total number of nodes

in the PSWs.

Table 5 Expreimental Data

PSW			Dfs.c	PM.c	Chory.c	Os.c
preprocessing		Size (Line of Code)	195	300	1065	1374
		Total # of nodes	147	254	817	1279
		# of global types	2	4	4	1
		# of global varriables	5	4	13	17
		# of functions	9	15	40	14
		# of clusters	14	20	51	17
		# of TVCC clusters	5	5	11	3
		# of function clusters	9	15	40	14
class extraction	attribute cluster	# of clusters	10~14	16~20	41~51	15~17
	method cluster	# of clusters	1~5	1~5	1~11	1~3
inheritance extraction	inheritance	# of new super classes	0	1	1	0

8. Conclusions and Future Research

This paper presented a methodology to extract classes and inheritance from PSW. The methodology was based on the idea of generating all possible groups of class candidates in combination and selecting a group with the best or optimal combination of candidates with respect to the degree of class group and hierarchy similarities. The methodology has innovative features: the usage of a deterministic clustering sequence method, the generation of all possible combinatorial cases of groups of multiple class candidates and inheritance based on abstraction and generalization, the usage of clustering methods as subjective decision rules based on the different types of the degree of interconnectivities for attributes and methods, the usage of an algorithm to measure both group and hierarchy similarities between multiple *n* class candidates in a group and *m* classes from a domain model, and the provision of statistics in these similarities of all combinatorial groups of possible candidates groups to a domain model to demonstrate diverse selection choice for the group of best or optimal candidates and inheritance. This methodology provides reengineering experts a comprehensive and integrated environment to select a group of the best or optimal class candidates.

The future research includes the persistence phase and the verification of equivalence between PSW and OOSW.

Acknowledgements

This work was supported by Basic Science Research Programs through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(2010-0023787), and the MISP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2015-H8501-15-1012) supervised by the IITP(Institute for Information & communications Technology Promotion), and Space Core Technology Development Program through the NRF(National Research Foundation of Korea) funded by the Ministry of Science, ICT and Future Planning(NRF-2014M1A3A3A02034792), and Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2015R1D1A3A01019282).

References

[1] Robert S. Arnold. *Software Reengineering*. IEEE Computer Society Press. 1994.

[2] G. Canfora, A. Cimitile and M. Munro. An Improved Algorithm for Identifying Object in Code. *Software-Practice and Experience*, Vol. 26(1), pp. 25-48. January, 1996.

[3] Doris L. Carver. Reverse Engineering Procedural Code for Object Recovery. *Proceedings of Conf. of Software Engineering & Knowledge Engineering*, pp. 442-449. 1996.

[4] P. Chen. The Entity-Relationship Model : Toward A unified View of Data. *ACM Transactions on Database System*, pp. 9-36. May 1976.

[5] Harald C. Gall, Rene R. Klosch and Roland T. Mittermier. *Architecture Transformation of Legacy System*. Technical Report Number CS95-418, Seattle. April, 1995.

[6] E. Horowitz. An Expensive view of reusable software. *IEEE Transaction on Software Engineering*, Vol. SE-10, No. 5, pp. 477-487. September, 1984.

[7] Yunsook Jin, Pyeong S. Mah and Gyusang Shin. Deriving an Object Model from Procedural Programs. *Proceedings of TOOLS97*. 1997.

[8] C. L. Ong and W. T. Tsai. Class and Object Extraction from Imperative. *Journal of Object-Oriented Programming*, pp. 58-68. March-April, 1993.

[9] Moonkun Lee, Changshin Jeoung and Myeongsun Jeong, A Reverse-Engineering Model using a Software Architecture. *Journal of KISS* (B), Vol. 25, No. 11, pp. 1630-1647. November, 1998.

[10] Panos E. Livadas and Theodore Johnson, A New Approach to Finding Objects in Programs. *Journal of Software Maintenance : Research and Practice*, Vol. 6, pp. 249-260. 1994.

[11] Dimitris N. Chorafas. *Cloud Computing Strategies*. CRC Press, 2010. ISNB: 978-1-4398-3453-4.

[12] Anca Daniela Ionita, Marin Litoiu & Grace Lewis, *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environment*. Information Service Reference, USA, 2013, ISBN978-1-4666-24887-7.

[13] S. Woo, J. On, M. Lee. An Abstraction Method for Mobility and Interaction in Process Algebra Using Behavior Ontology. *Computer Software and Applications Conference (COMPSAC)*, 2011 IEEE 35th Annual, pp.128-133.

[14] M. Lee, J. Choi. A Calculus for Transportation Systems. *Computer Software and Applications Conference (COMPSAC)/MVDA*, 2014 IEEE 38th Annual.