Object–Oriented Model Refinement Technique in Software Reengineering*

Wei-Jin Park, Sang-Yoon Min, and Doo-Hwan Bae
Department of Computer Science
Korea Advanced Institute of Science and Technology
373-1, Kusong-dong, Yusong-gu, Taejon 305-701, Korea
{wjpark,sang,bae}@salmosa.kaist.ac.kr

Pyeong-Soo Mah
Object-oriented Lab.
System Engineering Research Institute
1, Ueun-dong, Yusong-gu, Taejon 305-333, Korea
mah@seri.re.kr

Abstract

Software reengineering *for object-oriented rearchitecturing offers an exciting opportunity in migrating old legacy systems to evolvable systems in a disciplined manner. In the twofold-strategy software reengineering for object-oriented rearchitecturing, one of the problems to be solved is to derive a proper object–oriented model from the output of reverse engineering and the output of forward engineering. In general, the outputs of forward and reverse engineering can be inconsistent in their abstract levels, the amount of design information, naming conventions, and structures. In this paper, we present an Object–oriented model Refinement Technique(*ORT*) to build a final object model in the twofold-strategy software reengineering. We first organize the information gained from reverse engineering into* specification information tree*, and then compare the entities in the specification information tree with the information from forward engineering using* tree-structured data dictionary *to produce the final model. We demonstrate the usability of* ORT *by an example.*

1. Introduction

There are many legacy systems still in use even nowadays. These systems, developed many years ago, are usually large, unstructured and poorly understood. However, it is not desirable to throw them away and rebuild the whole system because such rebuilding would require so much cost in terms of time and effort, and the users of the systems would suffer from the temporal absence of the services during the development of new systems.

Software reengineering offers an exciting opportunity to resolve these problems, by offering an approach to migrating a legacy system to an evolvable system in a disciplined manner. *Software reengineering* is *the process of creating an abstract description of a system, reasoning about a change at a higher level of abstraction, and then re-implementing the system*[8]. In recent years, the object-oriented paradigm has been adopted in many software industries for developing new products. Accommodating with the trend, the software reengineering for object-oriented rearchitecturing has been gathering much attentions for reducing development costs and to facilitating the software maintenance.

Many reengineering research groups have reported that *domain knowledge* plays a critical role during the software reengineering process [1][2][4]. The domain knowledge can be helpfully used in the form of patterns of informal or semi-formal information such as a model, a program or the human individual himself. Such knowledge can be defined as *conceptual abstractions* [2]. Software reengineering process that contains conceptual abstractions as one of its inputs for the process of refining a model from reverse engineering is defined as *twofold-strategy software reengineering*[6]. In order to reflect domain knowledge to design information recovered from reverse engineering in the *twofold-strategy software reengineering*, a systematic refinement process is necessary to guide the process of reflecting the useful information from forward engineering into the information from reverse engineering.

In software reengineering for object–oriented rearchitecturing, domain knowledge is usually represented as an

*This work has been supported by System Engineering Research Institute

object model. During the refinement process, two object models from different origins, one from reverse engineering(*RooAM*, Reverse generated oo Application Model[4]) and the other from the domain knowledge(*FooAM*, Forward generated oo Application Model[4]), should be compared. Due to the fundamental difference in their origins which each model was built from, there are inherent differences in their abstraction levels, naming conventions, design information, and structures in the *RooAM* and the *FooAM*[1][4]. In order to build an optimal final object model out of these two models, effective comparison and refinement of the two models are essential. In this paper, we introduce a refinement technique that overcomes the previous mentioned differences in the two models, *RooAM and FooAM*.

We call this technique **O**bject–oriented model **R**efinement **T**echnique(**ORT**). The rest of this paper is organized as follows. In Section 2, we review related works including the concept of *twofold-strategy* software reengineering and its validity. In Section 3, we will explain in detail suggested data structures for **ORT**, their construction steps, and how they are used in **ORT**. Then we will present the process steps of **ORT**. Section 4 reports the preliminary results of **ORT** with an example. Finally, Section 5 discusses the proposed technique and the future work.

2. Related Works

In this section, we summarize the current major reengineering researches.

2.1. COREM

COREM[1] is a process and framework developed at Vienna Univ. of Technology for reuse-oriented reengineering[4]. COREM realizes the extraction of capsules similar to objects from an existing system implemented in a procedural language. In the COREM process, human intervention is naturally recommended for improving the quality of the object–oriented model developed through the reengineering process. COREM is considered as a representative framework with a twofold strategy and a reuse-oriented reverse engineering method for extracting reusable components from existing programs. Also the concept of *application-semantic* was defined first in COREM[4]. On the other hand, it does not address the issues encountered in the object refinement process systematically, such as differences in abstraction levels, naming conventions, amounts of design information and structures. In this paper, we focus on these problems, and propose an approach to resolve them. Figure 1 shows the overall framework of COREM.

[1]Capsule Oriented Reverse Engineering Method

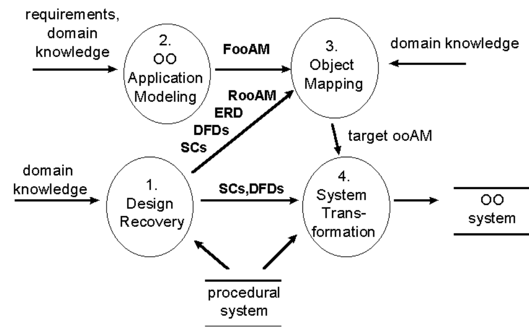

Figure 1. Overview of COREM framework

2.2. Rigi

Rigi is a system for analyzing evolving software systems through the reverse engineering technique developed at Univ. of Victoria[11]. The main goal of the Rigi system is to extract abstract information from software representations for software engineers in order to facilitate software evolution. Its focus is on summarizing, querying, representing, visualizing, and evaluating the structure of large, evolving software systems. Rigi has been used in the discovery, reconstruction, and evaluation of subsystem structures in existing software systems in the investigation of spatial and visual relationships among software artifacts for program understanding[7]. It has been proven to be successful during demonstrations at several software engineering conferences around the world. In this paper, we also use Rigi for the reverse engineering of our C source code example. Rigi 5.4.3 can be obtained from public domain.

3. Object–Oriented Model Refinement Technique

In this section, we explain **ORT** process in detail. In Section 3.1, we introduce the *specification information tree* suggested to resolve possible inconsistencies of abstraction levels between the two object models(*RooAM and FooAM*). In Section 3.2, we explain the *tree-structured data dictionary*, developed for resolving the naming inconsistencies between the two object models. In Section 3.3, we integrate these data structures into **ORT** steps, and explain all the steps in **ORT**.

3.1. Building a Specification Information Tree

The concept of information tree was originally invented for effective checking of completeness and consistency between given requirements statements and its object-oriented

requirements specifications(OORS)[12]. Information tree is built from formal specification written in a formal specification language. The *specification information tree*(SIT) is an extended form of the information tree to represent the *application-semantic*, while preserving the original properties of the information tree for completeness and consistency checking.

Among the data structures used in a software system, some of them are more important than others. Most of the data structures are used for local purposes and not crucial to understand the software globally. We believe that the data structures used globally among modules are more important than those used locally in terms of understanding the software under reengineering. Those important data structures are defined as *application-semantic*[4].

The following steps are performed in constructing an SIT :

step 1 Make class specifications from *RooAM*.

step 2 Create a root node that represents the whole system. This root node forms the level 0 of the tree.

step 3 Attach class nodes to the root node. Each class node corresponds to a class in the system. These class nodes form the level 1 of the tree. Each class is represented as a circle with the corresponding name for each.

step 4 In the level 2, attach method nodes to the corresponding class node. In this step, the special method, **main()** is drawn as a rectangle to distinguish main() from other methods.

step 5 In the level 3, attach the attribute nodes to the corresponding method node. Each attribute identified as an application-semantic through identification heuristics is denoted by a double circle. Child nodes of a rectangle node(**main()**) are also drawn as double circles because the distance between data structures and main() function can be a factor for application-semantic identification.

step 6 For each class node in level 1, examine if it has any double-circled nodes as its child in the level 3. Change the arc between such class node and root node into bold type. Such classes become application-semantic.

step 7 If there are constraint conditions for nodes in the level 3, they are represented in the level 4 with circles.

step 8 If any two classes in level 1 are communicate with each other, add dashed arcs between the classes in the level 1, by examing a call-graph or other documents that can be used for system's dynamic behavior[12].

Let $\mathcal{S}$ be the specification information tree for an object model. We can define various operations on $\mathcal{S}$ as follows :

1. $asDS(\mathcal{S})$: return a set of application semantic data structures in $\mathcal{S}$
2. $Uncertainties(\mathcal{S},e_i)$: If an entity(attribute or method) e_i in $\mathcal{S}$ has uncertainties for its membership, return all classes that it belongs to. It returns one class in case of no uncertainty. Uncertatinties mean the situation such that an entity are decided to belong into several classes due to the lack of design information. This can happed in case that the object-oriented requirements specification are generated from the reverse engineering.

Figure 5 in Section 4 shows an example of SIT. In the case shown in Figure 5, $asDS(\mathcal{S})$ is the set of $\{NUM, ID, Name, Month, Date, Read, pages\}$ and $Uncertainties(\mathcal{S},RetrieveTitles)$ is the set of $\{tagRECORD, ITEM\}$.

3.2. Building a Tree-structured Data Dictionary

In **ORT**, two object models, *RooAM* and *FooAM*, are used as the input for the process of building a final object model. We propose a *tree-structured data dictionary*, called TsDD, for combining data dictionaries of those two object models(*RooAM and FooAM*). The following steps are performed in constructing the TsDD :

step 1 Identify all entities appeared in *RooAM*.

step 2 List the identified entities with tag *C* for the class entities, *M* for the method entities, *A* for the attribute entities in the lexicographic order. Each entity, identified in this step, becomes root for each sub-tree. We do not give descriptions to each entity because we have little knowledge about the source code.

step 3 Underline entities that are identified as application-semantic.

step 4 Identify all entities appeared in *FooAM* with the same tag types mentioned in step 2.

step 5 Make each entity identified in step 4 a child of the lexicographically closed entity prepared in step 2. If there are many lexicographical-closed entities, map the entity identified in this step to the one of the candidates that has the same tag. Programmers are likely to name entities in the similar style that has similar semantics. For example, `person, perInfo, pers,`

`pItem` are used for people information. Thus, we arrange entities identified in **step 4**(from *FooAM*) near the lexicographically closed entities prepared in **step 2**. Eventually, many trees, that have entities in **step 2** as their roots, will be made.

step 6 Give a textual description to each child node(entities in step 4) in TsDD. It can be easily done because the leaf nodes originate from the domain knowledge.

step 7 If there are entities that have similar semantics between trees, connect these entities with *an arc*(i.e. `customer` **and** `client`**).**

step 8 Finally, group entities with the same prefix(postfix). This is due to the fact that entities with the same prefix(postfix) are likely to do functionalities in the same categories. For example, the function names that begin with `Comp` in one system, may have functionalities such as *computer something*.

An example of TsDD is given in Figure 2.

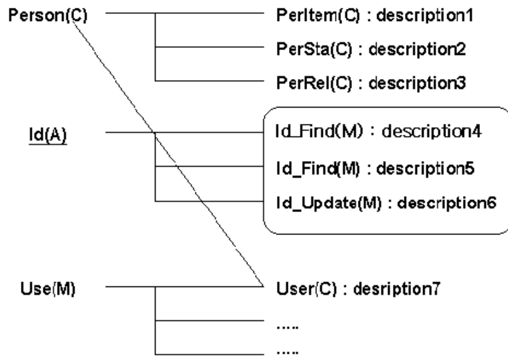

Figure 2. Example of TsDD

Let a TsDD generated from a SIT $\mathcal{S}$ and an object model $\mathcal{O}$ be $\mathcal{D}^{\mathcal{S}}_{\mathcal{O}}$. A root entity and A child entity in $\mathcal{D}^{\mathcal{S}}_{\mathcal{O}}$ are denoted by r_i, c_j, respectively. The followings are the operations defined on $\mathcal{D}^{\mathcal{S}}_{\mathcal{O}}$.

1. $Ren(r_i, c_j)$: Renaming entity r_i with c_j. Child entity c_j is based on domain knowledge and usually has semantics.

2. $Map(\mathcal{D}^{\mathcal{S}}_{\mathcal{O}}, r_i)$: Among children of r_i in $\mathcal{D}^{\mathcal{S}}_{\mathcal{O}}$, choose an entity that is capable to refine r_i, and return the entity. If there are several candidate entities that can be used to refine r_i, we combine those entities to make a new entity that has reasonable semantics, and return it. This decision can be made with the help of domain knowledge(*FooAM*), and the understanding of source code through SIT. To perform operation Map , one must know the characteristics of r_i. We can get such information by performing the operations, $Uncertainties$ and $asDS$ on $\mathcal{S}$.

3.3. Object–Oriented Model Refinement Procedure

ORT basically follows the similar style of object model building procedure in **OMT**[10]. However, while the object model building procedure in **OMT** is done on the basis of the knowledge from domain analysis, the building procedure for the object model in **ORT** considers both the domain knowledge and the information from the reverse engineering. **ORT** is performed using the following steps :

step 1 Build a SIT $\mathcal{S}$ from *RooAM* $\mathcal{O}_{\mathcal{R}}$.

step 2 Prepare an empty data dictionary from SIT. This step is the same as step 1 $\sim$ step 3 in TsDD building.

step 3 Identify objects and classes from problem statements.

step 4 Identify associations(and aggregations) between objects.

step 5 Identify attributes of objects and links. Through step 1 $\sim$ step 5, we can build an *FooAM* which could have similar abstraction level to $\mathcal{O}_{\mathcal{R}}$ with the help of SIT. Let an object model of such an *FooAM* be $\mathcal{O}_{\mathcal{F}'}$ comparing to general instance of *FooAM*. We call a general instance of *FooAM* as $\mathcal{O}_{\mathcal{F}}$. In COREM, domain knowledge is modeled as $\mathcal{O}_{\mathcal{F}}$.

step 6 Prepare a TsDD. This step is the same as step 4 $\sim$ step 8 in TsDD building procedure. Let a TsDD built from a SIT $\mathcal{S}$ and $\mathcal{O}_{\mathcal{F}'}$ be $\mathcal{D}^{\mathcal{S}}_{\mathcal{O}_{\mathcal{F}'}}$.

step 7 Compare two entities connected by an arc in $\mathcal{D}^{\mathcal{S}}_{\mathcal{O}_{\mathcal{F}'}}$. If there is an arc in the prepared TsDD, a comparison process can be started from that arc. The comparison process needs other documents from reverse engineering, such as a call graph, a data flow diagram(DFD), and a source code to understand the entities from $\mathcal{O}_{\mathcal{R}}$. In this step, the reengineer can reuse the components in $\mathcal{O}_{\mathcal{R}}$, employ new components from $\mathcal{O}_{\mathcal{F}'}$, or modify the two components, one in $\mathcal{O}_{\mathcal{R}}$ and the other in $\mathcal{O}_{\mathcal{F}'}$(new relationship etc.). Comparing process will make a final object model $\mathcal{O}_{\mathcal{ORT}}$.
Initially, $\mathcal{O}_{\mathcal{ORT}}$ is an empty set. Let r_i be a root in $\mathcal{D}^{\mathcal{S}}_{\mathcal{O}_{\mathcal{F}'}}$. In this step, We can get $\mathcal{O}_{\mathcal{ORT}}$ as follows :

$$\mathcal{O}_{\mathcal{ORT}} \equiv \mathcal{O}_{\mathcal{ORT}} \cup \bigcup_{i=1}^{n} Ren(r_i, Map(\mathcal{D}^{\mathcal{S}}_{\mathcal{O}_{\mathcal{F}'}}, r_i))$$
(n is the number of roots in $\mathcal{D}^{\mathcal{S}}_{\mathcal{O}_{\mathcal{F}'}}$)

step 8 Iterate and refine the model. There can be entities in $\mathcal{O}_{\mathcal{F}'}$ not corresponding to the entities in $\mathcal{O}_{\mathcal{R}}$. We can examine the classes that have relationships with the classes that were already mapped to $\mathcal{O}_{\mathcal{R}}$ and the related classes in $\mathcal{O}_{\mathcal{ORT}}$. This decision needs human intervention and will be infeasible to be fully automated. Figure 3 shows the execution flow of whole **ORT** steps.

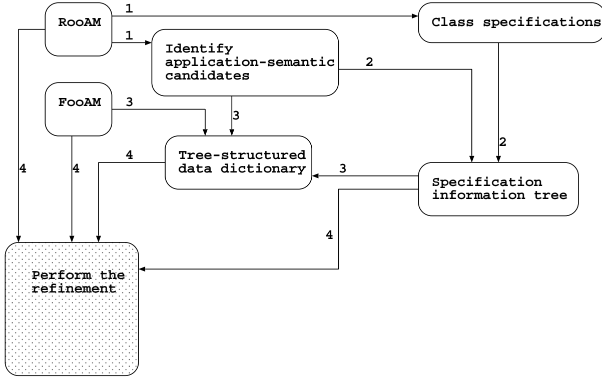

Figure 3. The execution flow of ORT

4. The Results of ORT

To explain and demonstrate **ORT** we proposed, a *BBS Analyzer* example is used. This kind of application is usually used among real BBS providers to find out the statistic information of their BBS system. The BBS analyzer chosen in this paper has statistic size shown in Table 1. For reverse engineering of the example program, we used `Rigi`[2] system, and we employed the type-based method[9]. Figure 4 shows the $\mathcal{O}_{\mathcal{R}}$ for BBS analyzer.

Metric	Value
Lines of Code	1325
Global variables	3
Functions	11
Types(user-defined)	4

Table 1. Statistic size of source code for BBS analyzer

In Table 2, it is shown that the results of applying the application-semantic heuristics to BBS analyzer. We

[2]We have used Rigi 5.4.3, the Rigi system circa 1996.

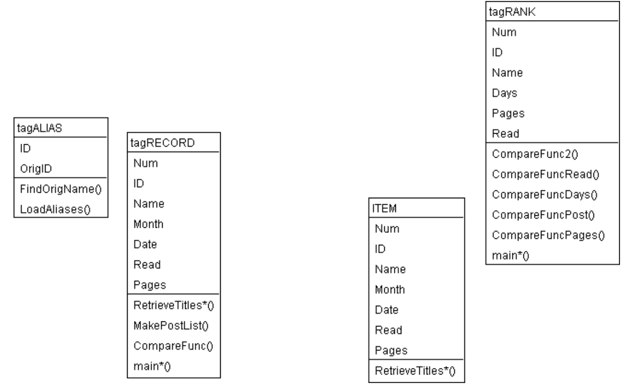

Figure 4. Object identified in BBS analyzer through reverse engineering

can find that *tagRank* structure plays an important role in the BBS analyzer. SIT for BBS analyzer is shown in Figure 5. By applying **ORT** steps for $\mathcal{O}_{\mathcal{F}'}$, we get an object model shown in Figure 6. Figure 7 shows a final refined object–oriented model($\mathcal{O}_{\mathcal{ORT}}$) and Table 3 and 4 show the refining procedures when applying **ORT**.

Heuristic	Variable list
Distance between ADT and main()	Rec(tagRecord) *Rank* *TempRank(tagRank)*
Pointer member in structure	Rec(tagRecord) *Rank* *TempRank(tagRank)*
Used in I/O function	*tempRank* *Rank(tagRank)*

Table 2. Application-semantic for BBS analyzer

In Table 3, italic entities are reused in $\mathcal{O}_{\mathcal{ORT}}$.

The refinement steps in COREM are focused on resolving uncertainties in $\mathcal{O}_{\mathcal{R}}$. Thus COREM refinement steps have little capability to change the $\mathcal{O}_{\mathcal{R}}$ itself. Figure 8 shows a final object model when COREM refinement steps were used. The uncertainties on methods *RetrieveTitles()* and *main()* can be resolved by domain knowledge. The new relationships between objects can also be made by domain knowledge.

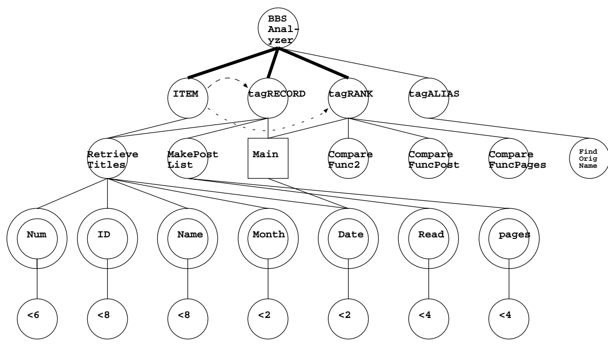

Figure 5. SIT for BBS analyzer

RooAM	*FooAM*	*ooAM*
Num	board_form	Num (board_form class)
ID	board_form	ID (board_form class)
Name	board_form	Name (board_form class)
Month	board_form	Month (board_form class)
Date	board_form	Date (board_form class)
Read	board_form	Read (board_form class)
Pages	board_form	Pages (board_form class)
ITEM, *tagRANK*		RANK_List (combined class)

Table 3. Mapping list for BBS analyzer

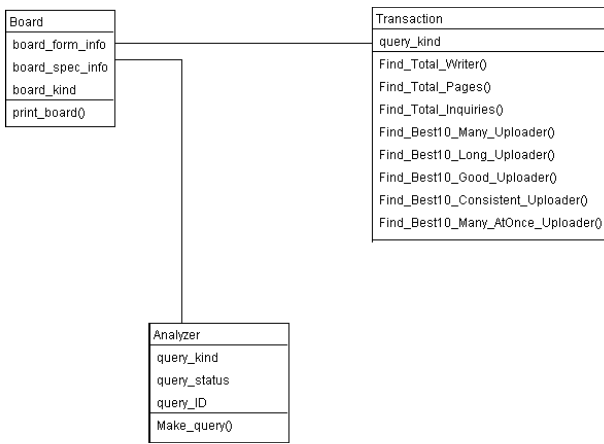

Figure 6. *FooAM* **for BBS analyzer**

Steps in **ORT**	Actions
8	Combine ITEM and tagRANK(RANK_List)
9	Identify the attribute board_form as a part of Board(aggregation)

Table 4. Changes of models in ORT

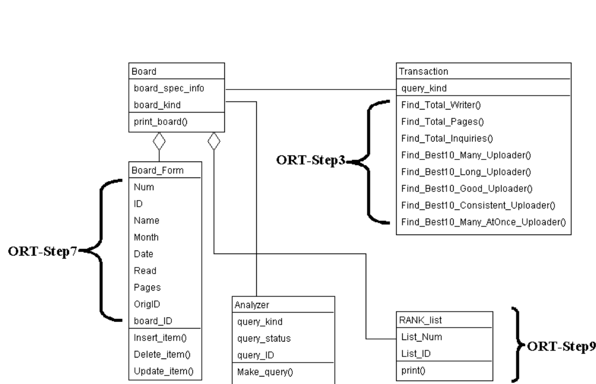

Figure 7. Object model $\mathcal{O}_{\mathcal{ORT}}$

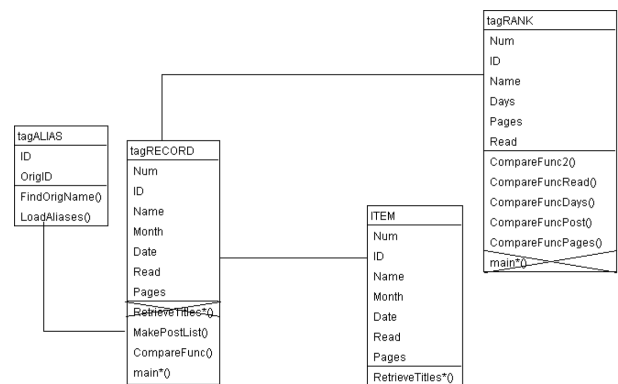

Figure 8. Object model $\mathcal{O}_{\mathcal{COREM}}$

5. Conclusions and Future Works

In this paper, we proposed **ORT** as a systematic object model refinement technique for the twofold-strategy software reengineering. As realized by many software engineering community, the human interventions required during the object-oriented reengineering process is one of the most critical obstacles. To reduce the amount of human interventions, and to guide engineers in more systematic manner, **ORT** provides the new refinement mechanism using SIT and TsDD that we have described so far. In addition, our approach has theoretical background in many perspectives, such as graph. We found that the productivity of *FooAM* played a critical role in the *twofold-strategy reengineering* because reengineering is quite different from redevelopment.

We have developed a graphic user interface supporting various document formats in **ORT**, and are currently in the process of integrating them together. We plan to perform more experiments on large-scale programs with the support of the **ORT** tool.

In order to support theoretical assurance of our approach, we have to study the verification steps for refinement process using SIT, in depth.

References

[1] T. Biggerstaff "Design Recovery for maintenance and reuse," *IEEE Computer*, 22(7):36-49,July 1990

[2] T. Biggerstaff "Human-Oriented Conceptual Abstractions in the Re-engineering of Software," *Proc. of 12th ICSE,* 1990, pp.120

[3] G. Murphy, D. Notkin "Reengineering with Reflexion Models:A Case Study", *IEEE Computer*, 30(8):29-36,August 1997

[4] H. Gall, R. Klösch "Capsule Oriented Reverse Engineering for Software Reuse", *Proc. of 4th European Software Engineering Conf.*, 1993, pp.418-433

[5] H. Gall and R. Klösch "Managing Uncertainty in an Object Recovery Process", *Proc. of International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*(IPMU'94), 1994, pp.1229-1235

[6] H. Gall, R. Klösch and R. Mittermeir "Application Patterns in Re-Engineering:Identifying and Using Reusable Concepts *Proc. of International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*(IPMU'94), 1996, pp.1099-1106

[7] H. Müller, M. Orgun, S. Tilley and J. Uhl. "A reverse engineering approach to subsystem structure identification," *Journal of Software Maintenance, 5(4):181-204,* 1993

[8] I. Jacobson and F.Lindstorm "Re-engineering of Old Systems to an Object-oriented architecture," *Proc. of OOPSLA91*, 1991, pp.77-83

[9] P. Livadas and T. Johnson "A New Approach to Finding Objects in Programs," *Journal of Software Maintenance, 6(2):249-260,* 1994

[10] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen *Object-Oriented Modeling and Design,* Prentice Hall International, 1991

[11] K. Wong *Rigi User's Manual Version 5.4.3,* Dept. of Computer Science, Univ. of Victoria, 1996

[12] S.Yau, D. Bae and K. Yeom, "An Approach to Object-oriented Requirements Verification in Software Development for Distributed Computing Systems," *Proc. of 18th COMPSAC94*, 1994, pp. 96-102