Université de Montréal

Migrating Legacy System towards Object Technology

par

Lei Wu

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Philosophie Doctor (Ph.D.)
en informatique

août, 2005

Université de Montréal

Faculté des études supérieures

Cette thèse intitulée :

Migrating Legacy System towards Object Technology

présentée par :

Lei Wu

a été évaluée par un jury composé des personnes suivantes :

Pierre Poulin
(président-rapporteur et représentant du doyen de la FES)

Houari A. Sahraoui
(directeur de recherche)

Petko Valtchev
(codirecteur)

Julie Vachon
(membre du jury)

Hafedh Mili
(examinateur externe)

Thèse acceptée le 20 octobre 2005

Sommaire

Durant la dernière décennie, les technologies orientées objet (OO) ont connu un grand succès dans le développement logiciel. Le paradigme OO facilite la conception et la compréhension de systèmes avec dissimulation et abstraction de l'information. Il est aussi muni d'une variété de dispositifs dont le développement et la maintenance de logiciel tirent grand profit. Comme résultat, les logiciels construits autour du paradigme OO sont en général plus réutilisables et plus faciles à maintenir. D'un autre côté, de nos jours, plusieurs logiciels hérités et vitaux qui ont été développés avant l'apparition de la technologie objet sont sévèrement confrontés à la difficile question de maintenance. La plupart de ces systèmes hérités subissent des changements continues pour satisfaire les nouveaux besoins. Ils présentent un haut niveau d'entropie: les systèmes hérités deviennent mal structurés, pauvrement documentés et faiblement modélisés. Afin de préserver la haute valeur économique de ces systèmes et du coup leur permettre de bénéficier des avantages de la technologie objet, leur migration aux technologies OO a été préconisée comme l'une des meilleures pratiques.

Bien que des méthodes de réingénierie qui visent la migration des systèmes hérités à la technologie OO aient été d'abord étudiées au début des années quatre-vingt-dix, la grande partie des travaux de recherche se concentrent seulement sur les techniques d'obtention des modèles objets du code du système hérité sans considérer en entier le processus de migration.

Pour résoudre ce problème, cette thèse présente un cadre de réingénierie pour aborder la problématique de migration des systèmes hérités. Nous avons exploré les éléments pertinents de l'évolution et de la réingénierie de ces systèmes, et avons développé un cadre d'application complet adapté à leur migration. Nous avons développé des outils et des techniques pour réduire la complexité et pour assurer la qualité du processus de migration. Ceux-ci incluent le développement de l'analyse des systèmes hérités, de la décomposition des logiciels, de la restauration de la conception, du modèle de restructuration de l'architecture, du modèle incrémental de migration et de

l'environnement de support. Les résultats expérimentaux illustrent l'efficacité et l'utilité de notre approche de migration.

Mots clés: réingénierie, rétro conception, systèmes hérités, migration, orienté objet, restructuration de l'architecture, modèle de migration.

Abstract

During the past decade, object-oriented technology has gained great success in software system development. It facilitates the design and understanding of a system with information hiding and abstraction. It also provides a variety of desirable features that greatly benefit to the practice of software development and maintenance. As a result, software systems implemented with the OO paradigm are in general more reusable and more maintainable. On the other hand, nowadays, many vital legacy software systems that were developed before the appearance of object technology are chronically facing the difficult question of maintenance. Most of these systems have undergone continuous changes to meet the evolution needs. They present a high level of entropy: the systems have become ill-structured, poorly documented, and weakly modeled. To preserve the high economic value of those legacy systems and meanwhile make them benefit from the advantages of object technology, migrating legacy systems towards object technology has been advocated as one of the best practices.

Although reengineering methods that aim at migrating legacy systems to object technology have first been studied in the nineties, most of the research work focuses only on the techniques to elicit object models from legacy code without considering the whole migration process.

To solve this problem, this thesis presents a reengineering framework to tackle the legacy migration issue as a whole. We have explored the pertinent issues concerning legacy system evolution and reengineering, and have developed a comprehensive framework to accommodate legacy system migration. We have developed tools and techniques to reduce the complexity and ensure the quality of the migration process. These include the development of legacy system analysis, software decomposition, design recovery, re-architecturing, incremental migration model, and supporting environment. The experimental results illustrate the effectiveness and usefulness of our legacy system migration approach.

Keywords: reverse engineering, re-engineering, legacy system migration, object-oriented, re-architecturing, migration model.

Acknowledgements

I would like to sincerely thank my research director professor Houari Sahraoui, for his profound supervision and continual support during all these years. His valuable guidance and pertinent advice have greatly helped me to conduct this research.

My earnest thanks would also go to my research co-director, professor Petko Valtchev. I genuinely thank him for his intensive help on my research, the numerous discussions on my research papers, and the rigorous reviewing of my thesis.

My thanks also go to Professor Yann-Gaël Guéhéneuc for many interesting discussions on the topics related to my thesis.

I thank my parents, my wife, and my sister for their unconditional support and consistent help for many years.

I also thank all the members in our GEODES group for their support and generous help.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

OO:	**Object Oriented**
UML:	**Unified Modeling Language**
LOC:	**Lines of Code**
AST:	**Abstract Syntax Tree**
CFG:	**Control Flow Graph**
CEG:	**Context Entity Graph**
CORBA:	**Common Object Request Broker Architecture**
OAP:	**Simple Object Access Protocol**
MU:	**Migration Unit**
***W*fMS:**	**Workflow Management System**
UDT:	**User Defined Data Type**
***SCC*:**	**Strongly Connected Components**
DA:	**Directed Acyclic Graph**
GA:	**Genetic algorithms**
GOAL:	**Genetic-based Object identification Algorithm**
C-COI:	**Conceptual Clustering for Object Identification**
OBC:	**Order-Based Clustering**
JNI:	**Java Native Interface**
API:	**Application Programming Interface**
CPM:	**Critic Path Method**
PERT:	**The Program Evaluation and Review Technique**
YAWL:	**Yet Another Workflow Language**
SGA:	**Simple Genetic Algorithm**

Chapter I. Introduction

1.1 Motivation

During the last three decades, a considerable amount of software was developed using procedural languages. For example, only the systems written in Cobol have been estimated to account for more than 100 billion LOC [Coy00]. Legacy software may be defined informally as an old software system that we do not know much about its design, but that is still performing a useful job. In many cases, it is critical to the operation of its owner [Ben95]. Although the term "legacy" implies something that is precious and inherited, in the domain of software engineering, this word nearly represents decision uncertainty: discard this old but mission critical software system or continue to evolve it with high maintenance cost. Nevertheless, ignoring the prevalence of legacy applications is difficult, and it is often not possible to replace or modify them easily [Luc97].

During the past decade, object-oriented technology (or simplified as object technology) has achieved great success in software system development. Object-oriented paradigm supports many modern programming methodologies, such as information hiding, inheritance, polymorphism, dynamic binding, etc. Object technology facilitates the design and understanding of a system with abstraction. It provides various desirable features that greatly benefit to the practice of software development and maintenance. As a result, software systems implemented in OO languages are in general more reusable and more maintainable. On the other hand, nowadays, many vital legacy software systems that wre developed before the appearance of object technology are chronically facing the difficult question of maintenance. Most of these systems are more than a decade old on average, designed only with the consideration of hardware limitations, and are

continuously modified to meet the evolutionary needs. Such legacy systems present a high level of entropy: the source code has become ill-structured, poorly self-documented, and weakly modeled. In addition, the documentation may present an inaccurate picture of what has been actually implemented [Ric97]. As a result, the high level of entropy combined with imprecise documentation about the design and architecture make their maintenance more difficult, time consuming, and costly. On the other hand, these systems have important economical values. A large amount of domain business knowledge has already been coded in legacy software. For the strategic assets they have preserved, it's impractical to discard them. Meanwhile, simply redesigning and redeveloping them by using modern technology is exposed to tremendous cost and risk. To preserve the high economic value of those legacy systems and meanwhile make them benefit from the advantages of object technology, migrating legacy systems towards object technology has been advocated as one of the best practices [Mel00]. This strategy focuses on leveraging existing legacy software assets while minimizing the risks involved in re-implementing large-scale mission-critical legacy applications [Uma97].

1.2 Problem Statement

In 1995, Bennett defined "legacy systems" as "large software systems that we don't know how to cope with but that are vital to our organization." [Ben95]. The Free On-Line Dictionary Of Computing (FOLDOC) defines as, "A computer system or application program which continues to be used because of the prohibitive cost of replacing or redesigning it and despite its poor competitiveness and compatibility with modern equivalents. The implication is that the system is large, monolithic and difficult to modify and evolve." [How02]. Sommerville defines it as "Older software systems that remain vital to an organization" [Som00]. After many years of maintenance, the quality of operation and maintainability has deteriorated dramatically due to many reasons, such as lack of up-to-date documentation, lost of key personnel, shift of technology of inter-operating peripheral systems, etc. Moreover, legacy systems usually consist of millions of lines of code, and a significant amount of business logic. Therefore, they represent an

important investment for their owners [Gam95]. From these, it's not difficult to find the characteristics of legacy systems:

- Old, using technologies that are now obsolete;
- Still perform crucial work for the organizations, and they represent a significant investment of their owners;
- Generally large, lacking of accurate documentations;
- Difficult to understand, hence hard to maintain.

1.2.1 Software Evolution

Informally, software evolution refers to all those improvement activities that take place after a software product has been delivered to the customer. A formal definition used by the Research Institute in Software Evolution at the University of Durham is: "The set of activities, both technical and managerial, that ensures that software continues to meet organizational and business objectives in a cost effective way". According to Bennett et al., evolution is a particular phase in the maintenance process, immediately after initial delivery, but before servicing phase out and close down [Ben00]. In reality, a software system will undergo maintenance throughout its life-cycle, such as correcting faults, improving performance, adapting to new environment, or adding new functionalities [Chi90]. Based on the empirical experiments on OS360 using a sequence of releases, Lehman firstly defined the laws of software evolution [Leh85]. His software evolution laws state that:

Law 1: "Software which is used in real-world environments must change or become less and less useful in that environment."

Law 2: "As an evolving program changes, its structure becomes more complex, unless active efforts are made to avoid this phenomenon."

From these, we can see that software evolution is a continuous progress of software fixing, adaptation, and enhancement.

1.2.2 Legacy Dilemma

Although legacy systems were implemented with old technology, they still provide important value by performing crucial work for their organizations, and usually they represent a significant investment and years of accumulated experience and knowledge [Bat98]. By applying structured programming technology, legacy systems soon meet the problem of continue evolution: structured programming is not designed to map onto entities in the real world. This makes it very hard to understand the original design when maintenance work is inevitable. The owner will eventually face what is called the legacy dilemma [Yan03][Ram99]:

- It is expensive and risky to replace the legacy system;
- It is even more expensive to maintain the legacy system.

Consequently, there is an urgent need to find ways to make legacy systems more maintainable without disrupting the operation of their owners.

1.2.3 Direction

In addition to improving legacy system quality, the reengineering process should also adopt new technologies to modernize/revive the target system, and ensure the efficient preservation of domain knowledge that has already been embedded in the legacy system. On the other hand, since most business activities typically involve multiple applications, this requires the integration of heterogeneous systems. To maximize the operational efficiency and leverage the value of existing systems, a suitable way to solve the legacy problem is to migrate into new open platforms. Moreover, sticking on aging or obsolete technology may cause maintainers eventually lose their enthusiasm [Tan98]. Most software engineers would rather work on new development projects which apply state-of-the-art technologies than maintain old applications that rely on obsolescent technologies – although this type of task also represents a different kind of challenge. As the research work of Tan and Gable [Tan98] illustrates that, compared with developers, there's a great tendency that maintainers will more easily depart their appropriate attitudes towards

maintaining legacy software which is based on aged technology. According to the legacy dilemma, neither replacing nor continuous maintaining legacy systems is acceptable. Therefore, migration toward new emerging technology is an appropriate direction.

Object-oriented systems can be designed and implemented in terms of artifacts that closely follow real world entities [Gam95]. It is advocated as a way to enhance software system's understandability, correctness, robustness, extendibility, and reusability, the key factors affecting software quality [Pre01]. Therefore, object technology makes it more natural for programmers to design and construct software models based on real world concepts. Additionally, the use of abstraction, encapsulation, inheritance, and other object-orientation techniques make object-oriented systems easier to understand and to maintain. Consequently, migrating an existing legacy procedural software into object-oriented paradigm is an appropriate approach to increase its understandability and maintainability [Jes99b]. Nevertheless, the migration itself is not an easy task. Extensive research has already been engaged in this field since the emergence of object technology [Jes99a]. With the steady progress of IT industry, the quick expansion of object technology in the software engineering domain has made it urgent to solve the "legacy dilemma".

1.3 Thesis Objectives

In this thesis, we propose a methodology and a set of techniques to assist software engineers migrating legacy systems towards object-oriented technology. We aim to improve the maintainability and reusability of target systems and facilitate the further evolution towards object technology. In particular, the objectives of our research can be summarized in the following points:

- Reveal the pertinent issues concerning legacy system evolution and reengineering.
- Provide a comprehensive methodology framework to migrate legacy system towards object technology.

- Develop tools and techniques to reduce the complexity and ensure the quality of the migration process. These include the development of legacy software decomposition, legacy design recovery, re-architecture, and techniques of identifying object models.
- Propose an incremental migration process model. The migration process thereby consists of a sequence of migration routes that progressively change the state of the system. The initial status corresponds to the original system and the final state corresponds to the target modernized system. The proposed re-engineering process aims at providing a comprehensive control mechanism by which optimized migration routes can be determined.
- Develop a migration supporting system. A collaborative migration project involves many people working together without the barrier of time and space differences. However, the large scale of collaboration in a typical migration project lacks of sufficient supporting techniques to facilitate project planning, monitoring distributive collaboration tasks, and communication. Our prototype of migration supporting system is designed to tackle these three important issues.

1.4 Thesis Overview

The remaining of this thesis is organized as follows:

Chapter 2: State of the Art: Literature Review

In this chapter, we will discuss the related research fields of our study topic in the Software Engineering domain. We will have a short literature review of the of related research.

Chapter 3: Migration Methodology Framework

In this chapter, we will present the blueprint of our migration methodology framework. It is constituted of four major components, namely legacy system analysis, Object-Oriented re-architecturing, incremental migration model, and

migration supporting system. The details for these four components will be presented in each of the following four chapters.

Chapter 4: Legacy System Analysis

In this chapter, we will present our legacy system analysis techniques. We focus on two directions, one is the legacy system decomposition, and the other one is the code collaboration pattern/role recovery and analysis.

Chapter 5: Object-Oriented Re-architecturing

In this chapter, we will discuss three major techniques that we have developed to conduct object modeling of legacy systems.

Chapter 6: Progressive Migration

In this chapter, we will present our progressive migration approach. The incremental migration process is designed to contain multiple iterations, each of which will only focus on migrating a certain part of the target system, leading to an increase in the portion of the renovated code and to a respective decrease of the legacy code. A fuzzy expert system is applied to prioritize the migration sequence.

Chapter 7: Migration Project Supporting System

The large scale of collaboration in a typical migration project lacks of sufficient support techniques to facilitate project planning, monitoring distributive collaboration tasks, and communication. In this chapter, we will present our approach and the prototype of *Caribou*: a supporting environment for migration project, to tackle these important issues.

Chapter 8: Experiment and Evaluation

In this chapter, we will discuss the experiments that we've conducted to migrate legacy systems into an object-oriented paradigm. We apply our migration approach and techniques to conduct these experiments. The results in turn validate our approach and also indicate potential improvement of our techniques.

Chapter 9: Conclusion

In this chapter, we will discuss our contributions and the directions of our future work.

Chapter II. State of the Art: Literature Review

In this chapter, we will discuss the related research fields of our study topic in the Software Engineering domain. Legacy migration is a research branch within legacy system maintenance and evolution. The driving force is the desired features provided by new technology and the inherent limitation of legacy systems. Reverse engineering and re-engineering are the major two fields that address different aspects of the legacy system migration problem. In this chapter, we will present a literature review of the recent advancements in these fields.

2.1 Reverse Engineering

Software reverse engineering is related to program property recovery as well as re-manipulation of system views. Chikofsky and Cross firstly introduced the taxonomy for reverse engineering and design recovery. They defined reverse engineering as "the process of analyzing a subject system to identify the system's components and their inter-relationships, and to create representations of the system in another form at higher levels of abstraction." [Chi90]. We should note that reverse engineering itself does not change the subject system or create a new system. It is a process of examination, not a process of alternation. Reverse engineering includes the following research areas [Chi90]:

Inventory/Analysis: applies source code analysis techniques and abstraction models to evaluate the technical, functional, and architectural aspects of the target system.

Positioning: focuses on restructuring the system to enhance its quality, or to improve source code understandability without changing the system external behavior.

Re-documentation: represents the system with a semantically-equivalent depiction in order to assist comprehension.

Design Recovery: aims at extracting meaningful higher-level abstractions of a system.

Many reverse engineering tools focus on extracting the structure of a legacy system with the goal of transferring this information in order to reengineer or reuse it [Mul00]. Over the past ten years, reverse engineering research has explored a wide range of fields in this domain, including subsystem decomposition [Bro95]Uma97], concept synthesis [Big94], program design pattern matching [Gam95][Ste98], program slicing [Tip95][Tak01][Rus02], analysis of static and dynamic dependencies [Sys99], object-oriented metrics [Chi94][Bey01], and software exploration and visualization [Pri93][Lud02]. In general, these kinds of analyses aim at addressing specific interests of program properties.

Reverse engineering research work generally follow a three step approach: 1) Facts extraction; 2) Abstraction, and 3) Presentation. Tilley et al. are among the first researchers to proposed the general framework of reverse engineering with the purpose of program understanding. They summarized these activities as "Data Gathering, Knowledge Organization and Information Exploration." [Til96]

2.1.1 Program Analysis

Jackson et al. defined program analysis as "The extraction of behavioral information from the software, represented as an abstract model or code" [Jac00]. It is the (automated) inspection of a program to infer its properties. In the past decade, various kinds of analysis techniques have been developed: static and dynamic, sound and unsound, operational and declarative [Jac00]. Extracting design models is the pivot for exploiting code analyses. Harandi and Ning firstly discussed the four levels of abstraction

for program analysis: implementation, structural, functional, and domain [Har90]. The implementation-level view examines individual programming constructs; the program is typically represented as an abstract syntax tree (AST), symbol table, or plain source text. The structural-level view examines the structural relationships among the program constructs; dependencies among program components are explicitly represented. The functional-level view examines the relationships between program structures and their behavior (function). The domain-level view examines concepts specific to the application domain. Program-analysis techniques may also consider source code in increasingly abstract forms [Not02][Nie99], including raw text, preprocessed text, lexical tokens, syntax trees, annotated abstract syntax trees with symbol tables, control/data flow graphs, program plans, and conceptual models.

Reasoning about code is another core activity of program analysis. Zeller concludes four well-known software analysis reasoning techniques [Zel03]: 1) Deductive program analysis: from an abstraction into the concrete—for instance, analyzing program code to deduce what can or cannot happen in concrete runs, such as static analysis generates findings without executing the program. 2) Observational program analysis: generates findings from a single execution of the program. 3) Inductive program analysis: summarizing multiple observations into an abstraction. It generates findings from multiple executions of the program. 4) Experimental program analysis: for isolating causes of given effects, e.g. narrowing down failure inducing circumstances by systematic tests.

2.1.2 Software Comprehension

The goal of software comprehension is to acquire sufficient knowledge about a software system so that it can evolve in a disciplined manner [Sco98]. During the process of software evolution, legacy systems have gone through years of maintenance. Changes have inevitably been applied to the source code, such as adding functions, fixing bugs, adapting with new environment, etc. [Chi90]. Meanwhile, many have failed to take a practical concern of keeping the documentation up-to-date [Ric97]. As the software

systme ages, the task of maintaining becomes more complex and more expensive. Poor design, unstructured programming methods, and crisis-driven maintenance can contribute to poor code quality, which in turn affects understanding. In many cases, the code is the only reliable source of information about the system [Bia00]. As a result, the process of reverse engineering has focused on understanding and analyzing the code.

Storey et al. defined program comprehension as "The task of building mental models of the underlying software at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for software maintenance, software evolution, and reengineering purposes." [Sto00]. It is a central activity during software maintenance: Corbi reports that up to 50% of the maintenance effort is spent on trying to understand code [Cor90]. According to Rugaber's general survey on program comprehension, the program understanding covers several important aspects including cognitive processes and automated techniques [Rug95].

The cognitive model of program comprehension models the mental processes involved in program understanding. In most cognitive models, hypothese are key drivers of the comprehension process. The models describe how programmers generate and verify such hypotheses [May95].

The other aspect of program comprehension focuses on deriving documentation from source code. The purpose is to re-generate the representation of a software system that allows programmers to gain a better understanding. This type of program analysis is called documentation generation. Deursen et al. listed four criteria for re-documentation [Deu99]: 1) Documentation should be available on different levels of abstraction; 2) Users must be able to move smoothly from one level of abstraction to another, without loosing their position in the documentation (zooming in or zooming out); 3) The different levels of abstraction must be meaningful for the intended documentation users; 4) The documentation needs to be consistent with the source code at all times. Here we list several representatives of systems re-documentation. DocGen is a documentation generation tool aimed at re-documenting legacy systems written in languages such as COBOL, DB2, JCL, and proprietary languages [Deu99]. The Rigi system can extract, navigate, analyze, and document the static structure of large software systems [Kie02]. NDoc is a source code documentation tool for the C# .net language [Nes02], and

Javadoc™ is a tool for generating API documentation in HTML format from doc comments in Java source code .

Most of the remaining aspects focus on techniques that help software engineers to understand target systems [Bal01]. Tilley summarized program understanding techniques into three categories [Til98]: 1) Behavioral approach, which emphasizes how the system works. It is top down and inductive, using a goal-driven method of hypothesis postulation and refinement based on expected artifacts derived from the knowledge of application domain. This approach begins with a pre-existing notion of the functionality of the system and proceeds to earmark individual components of the system responsible for specific tasks. 2) Functional approach, which relies more on the knowledge of the implementation domain to create abstract concepts that may map to the application domain and to the system's functional requirements. It is bottom up and deductive. This approach reconstructs the high-level design of a system, starting with source code, through a series of concept recovery steps. 3) Opportunistic, which combines top-down and bottom-up comprehension models to define how a software engineer understands a program. This opportunistic approach involves creating, verifying, and modifying hypotheses until the entire system can be explained using a consistent set of hypotheses.

2.1.3 Software Visualization

Reverse engineers also apply other means to facilitate the software understanding process. Software visualization is one of the most promising techniques. Charters et al define software visualization as "a discipline that makes use of various forms of imagery to provide insight, understanding and to reduce complexity of the existing system under consideration." [Cha02]. They emphasize that visualization provides a view of quick program comprehension that eliminates the overwhelming complexity of software systems. Stasko et al. describe software visualization as "the use of computer graphics and animation to help illustrate and present computer programs, processes, and algorithms." [Sta98]. Many visualization systems have been developed to assist program understanding. Here we list two of those most representative works. The SHriMP

visualization tools -- Simple Hierarchical Multi-Perspective views, use a nested-graph formalism and a fisheye-view to manipulate large graphs while providing context and preserving constraints such as orthogonality and proximity. SHriMP has been incorporated in the RigiSystem [Kie02]. CodeCrawler is a language-independent reverse engineering visualization tool for systems written in object oriented programming languages [Lan03]. It combines metrics and visualization to provide deep inspections of target system.

2.1.4 Architecture Recovery

Software architecture consists of the computational components, their interactions (connectors), and the constraints on them. Bass et al. define software architecture as "The structure of the computing system, which comprises software elements, the externally visible properties of those elements, and the relationships among them" [Bas03]. ANSI/IEEE Standard describes architecture as "Embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution" [Sta00]. The latter definition is intended to encompass a variety of uses of the term by recognizing their underlying common elements. The key point among these definitions is the need to understand and control those constructional elements in order to capture the system's utility.

Architecture recovery, also known as reconstruction [Sto03], is the attempt to extract software architecture from the source of the target system [Kaz99]. It is aimed at supporting the process of program understanding and reengineering for the purpose of software maintenance and evolution [Bri02]. It provides engineers with a global view of the system. This overview indicates the main components of the system, how they are related, and the constraints among them [Ede03]. During the past decade, many automatic techniques for architecture component recovery have been developed. Koschke proposed a framework to classify them based on a comparison of 23 techniques. He introduced the categorization of connection, metric, graph and concept based extraction approaches, and analyzed the commonalities and variability of these techniques [Kos02].

His research work shows that none of those automatic architecture component recovery techniques has the satisfactory precision. He further pointed out that semi-automatic techniques should be more suitable for the recovery work.

Architecture recovery primarily comprises two parts. One is the discovery of components (the computational portion); the other one is the detection of connectors (the interactions and communications among components). One major research topic in the first part is the detection of subsystems [Lak97], and the recovery of objects and abstract data types [Kui00]. Connector recovery is especially important for concurrent and distributed systems [Fiu96]. However, for most legacy systems, they are sequential and monolithic. Function call is the most primitive and dominating type of connector of such systems [Kos02].

Riva et al. developed an architecture extraction process and the environment supporting it [Riv00]. The process consists of four parts, which are applied iteratively in order to extract an increasingly refined view of a system's architecture. The four parts are: 1) Definition of architecturally significant concepts. 2) Data gathering, in which a model of a system is built in terms of the concepts defined in the first step. 3) Abstraction, in which the model is enriched with domain specific abstractions that lead to a higher view of the system. 4) Presentation of the rebuilt architecture in a series of formats, such as graphs, hyperlinks, and UML diagrams, representing the required architectural view-logical, process, physical and development.

Another interesting direction in this field is the detection of the architecture rationale. Deursen proposed a framework to recover the architecture rationale of architectural decisions [Deu01]. It includes three parts: 1) Use existing documentation, comments, and log messages as rationale pointers: System browsers aiming at architecture presentation can integrate these with other views. 2) Recognize design patterns, which aims at capturing specific rationale knowledge of recurring solutions. For procedural systems, design pattern recovery is complicated by the lack of an explicit pattern catalog. 3) Record rationale by combining architecture browsers with either simple annotation mechanisms or more involved rationale capturing tools. The recovery of architecture rationale can significantly support the understanding process.

2.2 Re-engineering

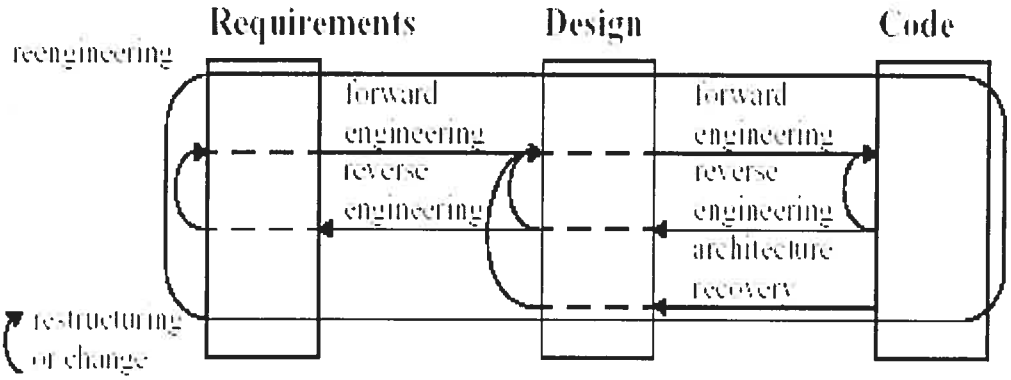


Figure 2.1: Relationships of Taxonomy Terms [Kos02].

Also known as renovation and reclamation, the "Reverse and Reengineering Taxonomy" defines re-engineering as "The examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form", whilst forward engineering is defined as "The process of moving from high level abstractions and logical, implementation-independent designs to the physical implementation of a system." [Chi90]. Koschke provides a visual representation of the relations among those terms as illustrated in Figure 2.1.

2.2.1 Restructuring

Software restructuring is the transformation from one software representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics) [Chi90]. It is aimed at improving program's quality while avoiding the modification of system functionality. Restructuring is often used as a form of preventive and perfective maintenance to improve the physical state of the subject system with respect to some preferred standard [Elo02]. A related term is refactoring, which is also a behavior preserving program renovation. However, it is a special case of restructuring, in the sense that it is always used in the context of

object-oriented programming, and typically at the level of programs (i.e., source code) [Fow99]. Restructuring can be more general, it can be applied to any kind of software artifact, in any language, at any level of abstraction [Men04]. Using Bowdidge's classification, software restructuring can be divided into four categories [Bow98]:

i. Scoping restructuring: alter the location where entities within a program are declared. An example could be moving a variable declaration out of a procedure, making it visible to other procedures in the system.

ii. Syntactic restructuring: alter specific characteristics of entities, such as converting compound statement structures into other, equivalent structures.

iii. Control restructuring: allow one to re-order the control structure within a program. This class of transformations not only requires control analysis, but also demands extensive data analysis.

iv. Abstraction restructuring: alter the data or code abstractions. It is often applied to restructure object-oriented systems. For example, it may be desirable to move a member from a child into its parent object to refine the inheritance relationship between objects. Code abstraction allows one to replace a statement, or sequence of statements with a procedure call.

A number of techniques have been proposed to implement restructuring. The main idea is to manipulate abstract program representations. Term rewriting, a technique used to specify a software system as a set of recursive equations, has been applied in the restructuring tools for COBOL systems [Bra98]. Griswold's tool for Scheme combines the abstract syntax tree (AST), control flow graph (CFG), and program dependence graph (PDG) [Gri91]. C_Structure is based on the AST and virtual control analysis [Mor98], and context entity graph (CEG) were developed to support specific language characteristics [Elo02].

2.2.2 Software Re-engineering

Reengineering is the analysis of existing software systems and modifying them to constitute in a new form. It seeks to clarify the understanding of software, alter the

characteristics of code, and improve the functionality/performance of the system. It is one of the research areas that aim at mastering software evolution. Actually, many factors such as business requirement changes, technological infrastructure shift etc. may all contribute to the constant demand for radical software change. Since the cost of system replacement is high, reengineering is hence often a better choice [Aeb97]. Software reengineering generally has three goals: 1) Enhancing system functionality; 2) Improving maintainability and reliability; 3) Migrating towards more advanced technology platforms. Since maintenance and change introduce new errors, the system's quality gradually deteriorates. Meanwhile, as systems grow, maintenance cost also increases. Therefore, one major objective of software reengineering is to re-design the system to improve system functionality and maintainability.

The reengineering process includes studying the system, making a specification at a higher abstraction level, adding a new functionality to this specification, and developing a completely new system based on the original one by using forward engineering techniques. Byrne has proposed a process model for reengineering with three parts: Reverse engineering, Alteration, Forward engineering, which he visualized as illustrated in Figure 2.2.

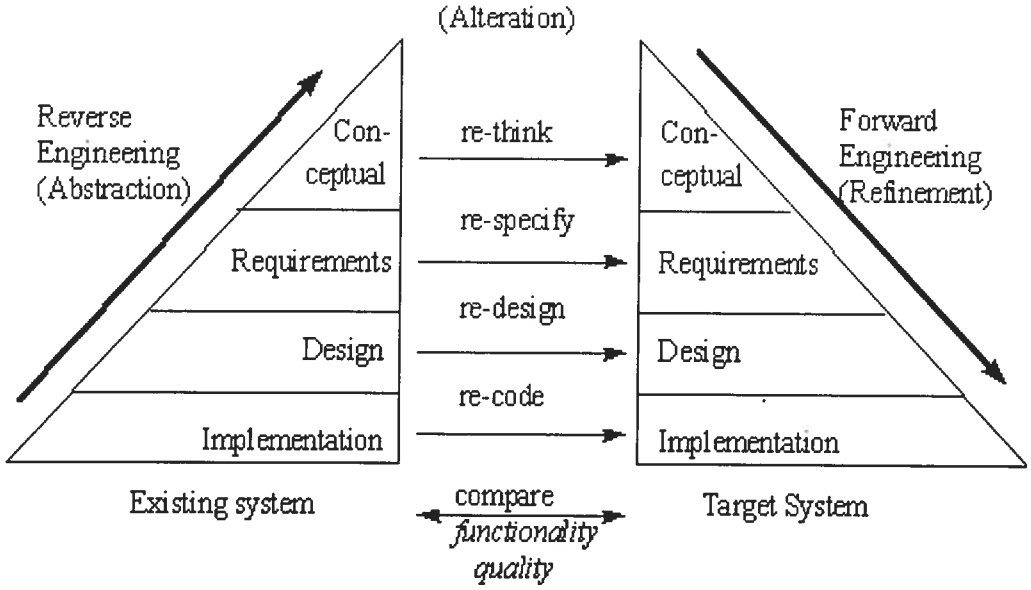


Figure 2.2: Re-engineering Model [Byr92].

The SEI Horse-shoe Model (see Figure 2.3) is similar to Byrne's model, but it takes a software architecture perspective on reengineering. Horse-shoe model aims to integrate the code-level and the architectural-level reengineering views. In its most fundamental form, there are also three basic reengineering processes [Car99]: i) Architecture recovery: the analysis of existing systems and the extraction of high-level abstraction artifacts from source code. This recovered architecture is analyzed to validate the conformance with the "as-designed" architecture. The discovered architecture is also evaluated with respect to a number of quality attributes such as performance, modifiability, security, and reliability. ii) Architecture transformation: the logical transformation of the high level abstractions. The "as-built" architecture is recovered and then reengineered to a desirable new architecture. It is re-evaluated against the system's quality goals and subject to other organizational and economic constraints. iii) Architecture-based development: the implementation of the desired architecture. Code-level artifacts from the legacy system are often wrapped or rewritten in order to fit into this new architecture.

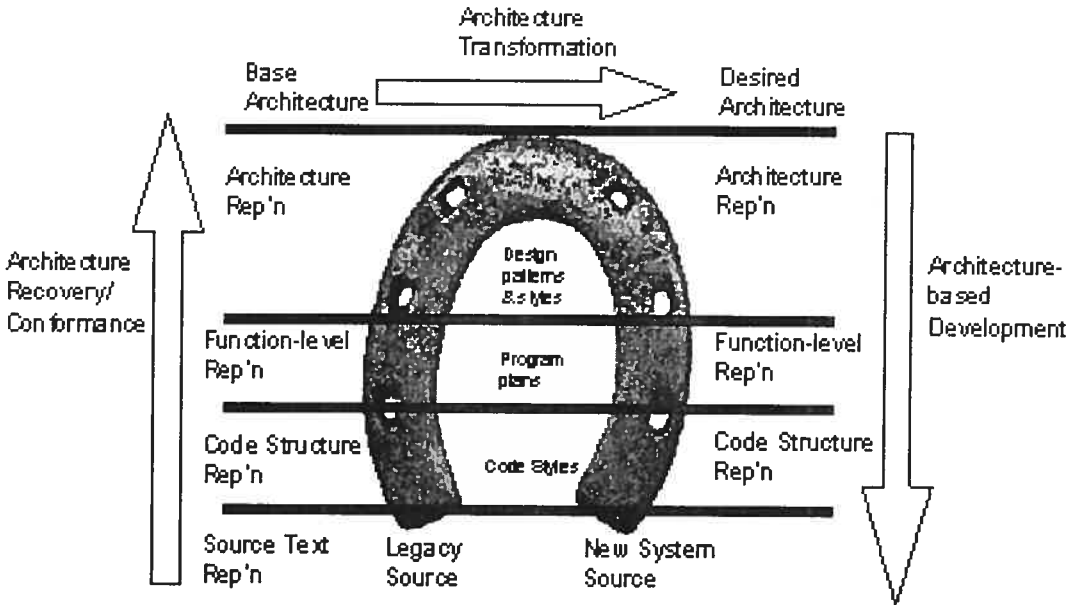


Figure 2.3: SEI Horse-shoe Reengineering Model [Ber00].

2.3 Migration towards Object Technology

With the constant development of object-oriented technology, migrating legacy software systems into object-oriented platforms has gained significant attention over the past few years [Luc00] [Zou01] [Mar02]. This research aims to increase legacy system's maintainability, reusability, and overall quality by means of reengineering it into an advanced paradigm, namely object oriented platform. By this way, the migrated legacy system may largely benefit from the advantages of object technology.

Legacy system migration encompasses many research topics. It addresses issues of reverse engineering, reengineering, object identification, schema mapping/translation, data transformation, human computer-interaction, testing, migration process control, etc. In the literature, migrating legacy software towards object technology can be further divided into two directions: the migration methodology, and the object identification technique. For the former one, the focus is on developing a general migration solution; the latter one is dedicated to building up a particular technique to conquer the object modeling issue in the migration process. The majority of the published research is concentrated on the latter direction.

Martin and Muller applied the Ephedra prototype to transform C source code into Java programs [Mar02]. Their approach includes three steps: (i) insertion of C function prototypes, (ii) data type and type cast analysis, and (iii) source code transformation. They aim at translating parts of C code into Java platform automatically, thus to avoid an entire redevelopment of business logic that has already been embedded in the existing system. Nevertheless, Ephedra has two major weaknesses in fulfilling its automation goal. One lies in its lack of concrete methodology to identify candidates for classes and their members; the other one lies in that it is hard to apply their method in fully automating the process of translating completely different languages. The difficulty exists in the syntax and semantic differences.

Gonzalez et al. proposed another approach to reengineer legacy procedural software systems into object-oriented technology [Gon98]. It includes two major processes: translation and transformation. In translation, the source code artifacts written in the original language will be substituted by the target language's artifacts without significant

modifications to legacy software architecture; transformation is a process that changes the original architecture into a new object-oriented architecture. In this approach, a procedural program is viewed as a poorly designed OO program with a single large class. The goal is to decentralize this large class into many smaller classes to carry out OO design. Park et al. further proposed an Object-oriented model Refinement Technique (ORT) to build a final object model [Par98]. It first arranges the information acquired from a reverse engineering process into a specification information tree; then it applies a tree-structured data dictionary to compare the entities in the specification information tree with the information from forward engineering, thus to finally produce the refined model. However, in this approach, deriving a proper OO model from the output of reverse engineering and forward engineering still remains a problem. This is mainly because the amount of design information, naming conventions, and structures of these two processes may be inconsistent.

2.3.1 Migration Forms

To migrate a legacy system towards object technology, there are several forms suitable for a certain migration project (see Figure 2.4). We summarize them as following.

- *The Pure Language level Transformation.* A significant effort is put on the issue of syntax and semantic swaps between different languages [And00] [Mar02], especially those having nearly the same semantic base [Fan99], such as C and C++. The benefit of this kind of research work is to facilitate the automation of the migration process at the pure language conversion level.

- *Migrating legacy system towards distributed net-centric OO computing environment.* This is to migrate standalone legacy systems into a distributed object computing model to fit with the net-centric (such as internet) environment [Li00] [Luc00]. Another direction is towards web computing platform. With the rapid growth of e-commerce, there's a large demand of migrating legacy information systems into web platforms. One approach is migrating the computation model from standalone computing model into web-based OO

computing model [Pat99] [Zou01]. By providing a framework [Zou01], this approach allows the identification of reusable business logics from legacy system in the form of legacy components, wrapping them into CORBA components to enable remote access, and finally translating into SOAP (simple object access protocol) to enable the internet application. The other direction is migrating the interface of legacy systems into web-based user interfaces [Mel00].

- *Migrating legacy information system (Database centric) towards modern object oriented database platform.* A large portion of legacy systems are data centric applications which mainly depend on the database management system. To modernize this type of legacy systems, the migration of data and host DBMS into an object-oriented database is the key issue [Jes99b] [Bia00]. They use a data-centric approach to migrate the data table from a legacy database system into an Object-Oriented database system. They use the new database system to construct the OO application.

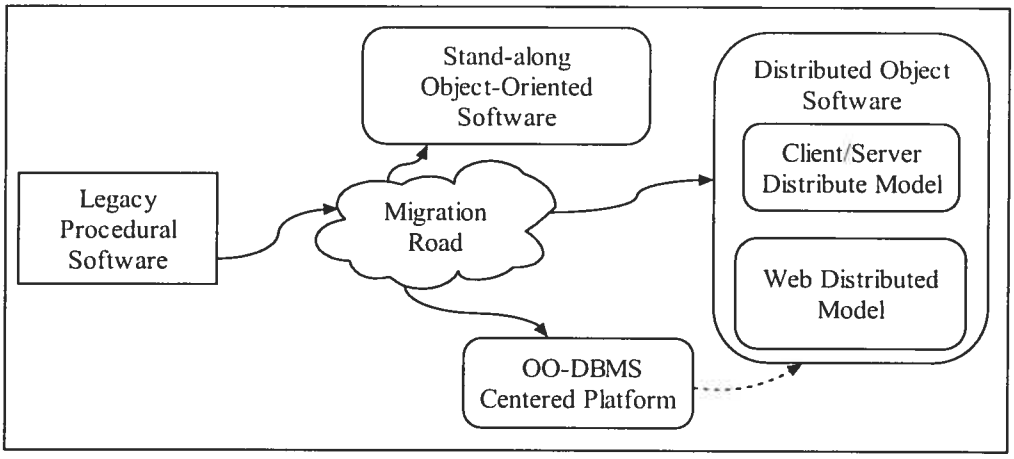


Figure 2.4: The Forms of Legacy System OO Migration

2.3.2 COREM Approach

COREM: Gall et al. proposed an expertise centered total migration framework [Gal95]. This approach aims at providing a sequential re-engineering model. The

migration process is well defined as a sequence of a bunch of activities together with their sub-objectives (see Figure 2.5). Activities are performed separately; each will deal with a particular issue of legacy migration. No partial object-oriented result is available until all the activities are finished. It's a natural method since most reengineering activities can be modeled in a sequential way, thus facilitating the organization of a linear migration process.

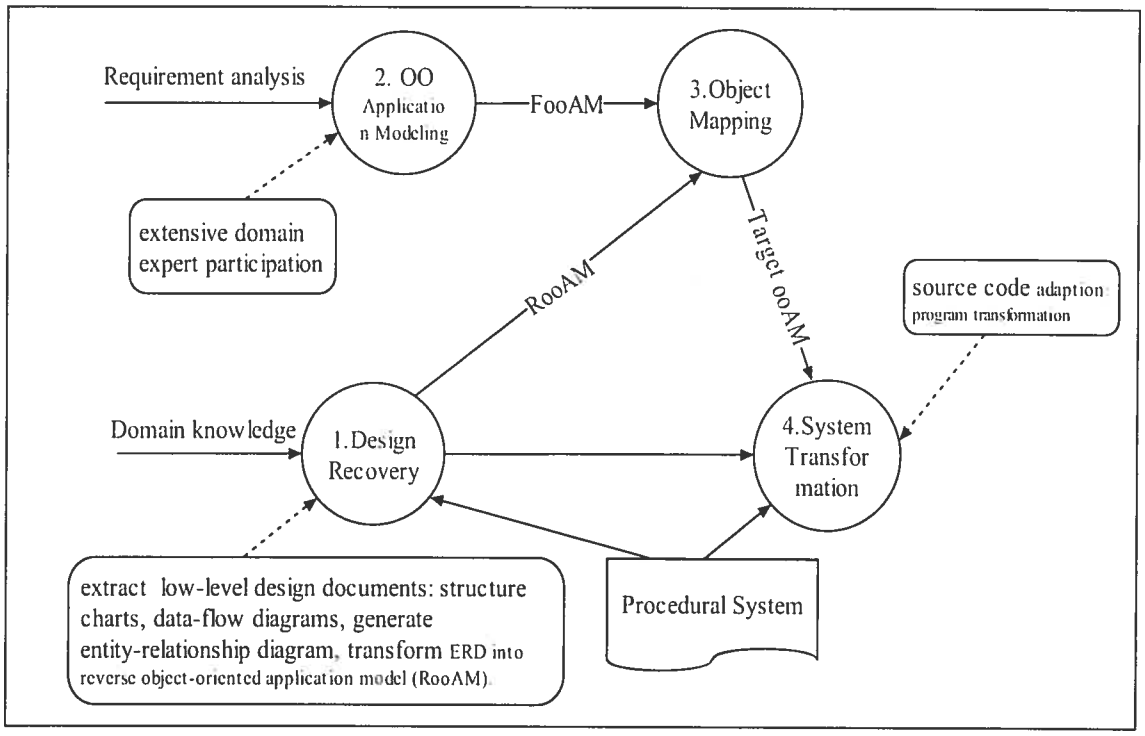


Figure 2.5: the COREM migration approach

The migration process is divided into several major parts, domain human experts are extensively used to help the understanding of legacy architecture and the elicitation of object-oriented models. The legacy system migration process consists of four main steps:

i. *Design Recovery,* in this step different low-level design documents (i.e., structure charts, dataflow diagrams), an entity-relationship diagram, and an object-oriented application models (called *reversely generated object-oriented application model,* RooAM) are generated from the source code of the procedural program.

ii. *Application Modeling,* based on the requirements analysis of the procedural input program, an object-oriented application model (called *forward generated object-oriented application model*, FooAM) is generated, working as input for the following object-mapping process.

iii. *Object Mapping,* in this step the elements of the RooAM are mapped to the elements of the FooAM resulting in a target application model (target ooAM). The target ooAM represents the desired object-oriented architecture and is used for performing the needed source-code adaptations.

iv. *Source-Code Adaptation,* the source code adaptation step completes the program transformation process on the source code level and is based upon the results of the previous steps, especially the target ooAM.

The major advantage of this type of migration method lies on that both the legacy system and the target object-oriented system will be treated as a whole, every aspect will be considered at the same time. Therefore, it will have a more integrated view of any migration stage, and reduce the communication work which is inevitable for other approaches between the target system and the subject legacy system.

The shortage is also obvious. To compensate for the advantage it brings, the resource consuming is enormous since every kind of migration activity is conducted in a parallel manner.

2.3.3 The ERCOLE Approach

ERCOLE -- Encapsulation, Reengineering, and Coexistence of Object with Legacy. In this research, Lucia et al. proposed an approach using a wrapping technique to migrate legacy systems written in the procedural language RPG into object-oriented platform [Luc97][Luc00]. The architecture is illustrated in Figure 2.6.

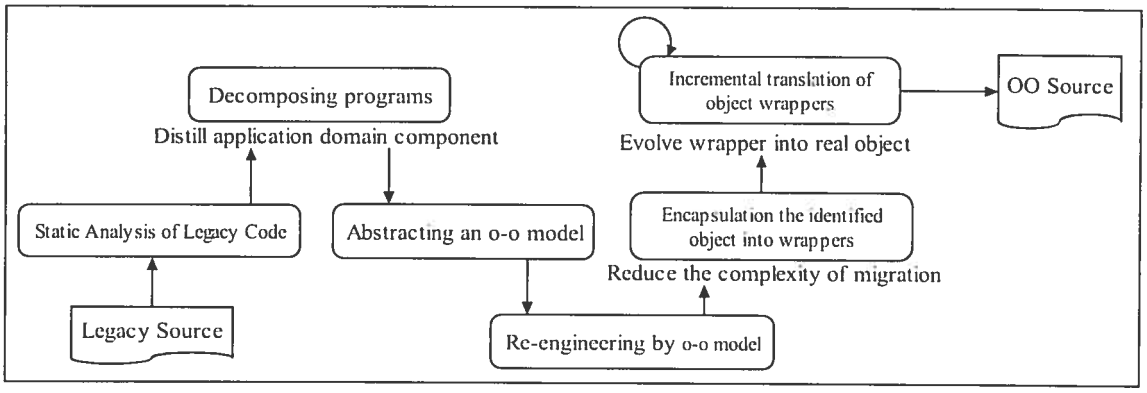


Figure 2.6: the ERCOLE legacy wrapping process

A systematic global migration approach based on a wrapping technique was well defined in this research. It contains six phases [Luc97] forming the integrated process to gradually migrate legacy code into an object-oriented platform:

1. *Static analysis of legacy code:* extract relevant information from source code and store it into a repository. Both fine/coarse-grained information at intra/inter-procedural level are recovered.

2. *Decomposing non-batch programs:* decompose the interface components and the application domain components.

3. *Abstracting an object-oriented model:* coarse-grained data-centered entities will be treated as object candidates, and look chunks as candidate methods. Chunks can be whole batch of programs, subroutines, or groups of related subroutines based on call graph or program slice.

4. *Re-engineering the system according to the results of decomposition and abstraction:* it's a modification of each divided source code parts, therefore to make them more easily to be encapsulated in the next stage.

5. *Encapsulating the identified objects into wrappers:* in this stage, the new object-oriented parts will be able to coexist within the legacy system. The legacy system or parts of it will be encapsulated into an object wrapper and executed on the original platform. The wrapper provides the interface through which the new object-oriented system can exploit existing resources.

6. *Incremental translation of object wrappers:* wrapper is a preliminary stage of the migration process, thus further object-oriented translation of wrapper will be executed, and different object wrapper will be able to migrate separately.

The kernel part of this approach is the application of wrapping legacy code into an object model during the migration process. Consequently, the original code is essentially entombed and integrated into the new system.

The advantage of this approach lies in three aspects: one is that legacy code becomes part of the new generation of the target system without discarding the value of the legacy system; the other one is that the construction of an object wrapper is relatively easier and more rapid than constructing a totally new object entity, it will require less understanding of legacy code thus reduce the implementation time. However, this is also the origin of its weakness: the quality of the target system will not be comparable to the pure object-oriented one. Finally, wrapping approach enables incremental migration which benefits a lot from reducing risk and cost by progressively conquer the problem.

The shortage also lies in three aspects. First, legacy systems, which have evolved over many years, are normally difficult to decompose. Therefore, in many cases, applying legacy wrapper technique is quite difficult. Second, adding wrapping code to encapsulate legacy components increases the migration complexity. Third, extra wrapper communication burden decreases the execution performance, thus reduces the quality of the target system. In a word, wrapping is a compromise approach.

2.3.4 Legacy Wrapping

Wrapping is a method of encapsulation that provides well-defined interfaces to access legacy systems [Sne98]. The major part of the legacy wrapping approach is encapsulating certain legacy code fragments into a component-like software entity. Such an entity will be designed with an object interface through which the objects can interact with each other or with the rest of the legacy system. It is a realistic approach, since it is

accomplished easily and rapidly with current available legacy code. Wrapper object makes new systems able to exploit existing resources of legacy systems, thus allowing an incremental and selective translation of the identified objects [Luc97]. To use wrapping, application developers must implement the interface towards subject legacy systems [Luc00]. Since there are many styles of interfaces between subject and target systems, the communication layer between legacy code and target code is the primary concern. There are two directions of legacy wrapping:

- Using a wrapping entity at an intermediate stage of migration process. The ultimate target system will be derived from the wrapped components. The encapsulated object entity in fact is not a result of real object-oriented design, it is an interim stage that works as a pseudo-object. During the migration process, a wrapping component plays a role as an object meta-model and further object derivation will be performed later.

- Using a wrapper as a final result. Thus the target system will use it as an object-oriented component part in the target system, especially in the distributed computing environment [Li00][Kim00].

The process of wrapping involves different techniques depending on the accessible elements of a legacy system. Ideally, the legacy system has a clean API and well documented services. It is then possible to define a direct wrapper interface that should be minimally affected when any change occurs [Li00].

Since each legacy system presents a unique constrained entity, it may have no APIs at all, or a limited API, or an extensive but proprietary API. Similarly, the legacy system may use sockets, RPCs, files, events, or any other number of message passing and inter-process communication mechanisms. The object wrapper therefore hides these idiosyncrasies and presents an interface that is consistent with the desired target software architecture.

The wrapping technique can be further classified into five different levels, namely, job process level, transaction level, program level, module level, and procedure level [Sne98]. At the process level and the transaction level, wrappers encapsulate a batch of legacy executive processes. Legacy applications are invoked through wrappers by creating the requested new process and directly deploying the corresponding service

without prior knowledge of the corresponding legacy code [Kim00]. At the module level and the procedure level, encapsulation focuses on clear interfaces, restructured and re-modularized legacy code. An object wrapper acts as a layer that maps one form of application program interface (API) to another. The layering can be done without modifying the underlying API design. The functionality of the layer interface depends on the existing underlying legacy system. Meanwhile, additional functionality can be added to the legacy system by enhancing the wrapper layer [Cim98]. Object wrappers can be categorized into following types: i) wrappers for legacy data; ii) wrappers for legacy functionality; iii) wrappers for legacy presentation logic modules.

Wrapping Legacy Data. Wrapping involves the addition of "layering" code to provide transparent access to legacy databases, relational databases, and flat files [Can99]. It encapsulates the data elements from legacy database or flat file entities as objects and provides interpretation mechanisms between legacy data resource requests and object APIs.

Wrapping Legacy Functionality. Object wrappers encapsulate parts of legacy code that implements specific legacy system functionalities. The encapsulation can be arranged at different levels of abstractions, including process level, transaction level, program level, module level, and procedure level [Sne98][Li00]. Encapsulation separates the interface from implementation. It can then be used to partition and modularize a monolithic legacy system [Kim00]. Each component can be encapsulated separately by the common interface, and then it can be reintegrated using an integrated communication mechanism used by the target OO system.

Wrapping Legacy Presentation Logic. In some cases, object wrapping of legacy modules related to the presentation logic may be the only option available to integrate legacy applications that are very old and non-decomposable [Li00]. Basically, the object wrapper is a layer on top of the screen scraper, which allows client applications to simulate the terminal keyboard/display features and thus acts as programmable terminal emulators. The object wrapper must include interfaces for all information that the user

actually types during a terminal session (login ID, password, key strokes, order number, etc.).

The legacy wrapping technique has several distinctive advantages:

- Wrappers provide access to a legacy system through abstract application program interfaces (APIs), regardless of the internal implementation complexity of the legacy system.
- Wrappers provide technology migration paths for legacy systems, allowing component upgrade without affecting the rest of the system.
- Wrappers eliminate the need to move the existing software out of its native operating environment.

The disadvantages are also very noticeable:

- If the wrapping system is the final result, then the migrated target system is not a full object-oriented system. The low-level implementation parts are still not re-engineered. Therefore, it hinders the migrant system from benefiting the advantages of object-oriented programming.
- If wrapping is only the middle status of the migration process, then it causes extra complexities in constructing wrappers, and co-operating legacy code with object wrappers.

2.4 Legacy Code Modeling Techniques

One of the key research issues of migrating legacy procedural program into an object-oriented paradigm is to identify object-oriented features from procedural code. Object identification can be looked as viewing existing systems in an object-oriented manner. This forms an active research direction of legacy system migration. In the literature, many modeling techniques are proposed to discover objects inside of legacy source code [New95][Sne95][Sif97][Lak97][Sah97][Cim99][Sne00][Can01][Kos02]. Some of these techniques can be partially automated, therefore greatly reducing the time of constructing object-oriented model from legacy code.

2.4.1 Similarity Clustering

The similarity clustering approach compares pairs of entities by their direct relationships in order to decide whether they belong to the same atomic component/object [Man98]. This technique groups base entities (subprograms, user-defined types, and global variables) according to the proportion of features (entities they access, their name, the file where they are defined, etc.) they have in common. The intuition is that if these features reflect the correct direct and indirect relationships between these entities, then entities, which have the most similar relationships, should belong to the same atomic component [Rai00]. The key issue for applying this technique is finding out for a certain aspect, how similar the program entities are. Subprograms are clustered into modules based on similarity metrics [Sch91]. Since the two most similar groups are combined per iteration, the order of combinations can be represented by a binary tree, in which the leaves are the initial groups and the inner nodes are combinations of groups. The farther a combination is away from the root of the tree, the higher is its degree of similarity. This procedure is called *hierarchical clustering* (see table 2.1).

```
Place each entity in a group by itself;
Repeat
        Identify the most similar groups Si and Sj;
        Combine Si and Sj;
        Add a subtree with children Si and Sj to the clustering tree;
Until the existing groups are satisfactory or only one group is left;
```

Table 2.1: Hierarchical similarity clustering algorithm

Clustering Criterion. In each iteration, the most similar groups are combined using the similarity metric. Many aspects can be considered to compare the similarity of two program entities, such as using the same user-defined data type, accessing the same files out side of the program, etc.

Similarity between Subprograms. The group similarity used to combine groups in the algorithm is based on a similarity between subprograms [Sch91]. Given two subprograms *A* and *B*, the similarity metric used during clustering is defined as:

$$Sim(A,B) = \frac{Common(A,B) + k * Linked(A,B)}{n + Common(A,B) + d * Distinct(A,B)} \quad (2.1)$$

where *Common(A,B)* reflects the common features of *A* and *B* and *Distinct(A,B)* reflects the distinct features. *Linked (A, B)* is 1 if *A* calls *B* or *B* calls *A*, otherwise it is 0. The two parameters $k >= 0$ and $d >= 0$ are weights given to *Linked* and *Distinct* in *Sim*. They have to be ascertained by experiments on a sample of the subject system. The parameter $n >= 0$ is used for normalization purposes. Features of a subprogram *A* are all non-local names that *A* uses including the names of procedures, macros, type-defines, objects, and even the individual record component names of structured types and objects. *Common* and *Distinct* are computed as weighted sums (*features(A)* denotes the features of *A*):

$$\begin{aligned} Common(A,B) &= W(features(A) \cap W(features(B)) \\ Distinct(A,B) &= W(features(A)/features(B)) + W(features(B)/features(A)) \end{aligned} \quad (2.2)$$

The term *features(X)* refers to the set of features of *X*, *W(X)* is the weighted sum of these features:

$$W(X) = \sum weight(x) \qquad x \in X \quad (2.3)$$

where $weight(x) >= 0$ is a weighting factor which allows assigning certain features more influence on the global value of the metric.

2.4.2 Concept Analysis

The use of concept analysis has been applied as an automated technique for analyzing the modular structure of legacy software [Sah99][Pao01]. Concept analysis is a

mathematical technique that provides a way to identify groupings of *items* that have common features [Gan99]. The main application is to derive the class structure of legacy software [Lin97][Sif97]. It starts with a *context*: a binary table (relations) indicating the features of a given set of items. From that table, the analysis builds up so-called concepts which are maximal sets of items sharing certain features. The relations between all possible concepts in a binary relation can be given using a concise lattice representation. The object identification is done by deriving a concept lattice from the code based on data usages in source code procedures. The structure of this lattice reveals the modularization (object class) that is implicated in the code.

Concept analysis has a sound mathematical background and the insights into the relationships among system components. It is an interesting technique for atomic component detection. On the other hand, it's also a time-consuming process when applying concept analysis to larger systems. This is the major barrier to apply concept analysis technique in object modeling task. Valtchev et al. have proposed a novel technique to build concept lattices from partitions [Val01], thus largely reducing the complexity to build a full-scale concept lattice.

2.4.3 Global Variables and Types

This technique focuses on the analysis of global variables, aggregate data types, and formal parameter lists. In the original legacy source code, global variables, data types, and their appearance list in formal parameter become primary candidates for classes in the new object oriented system. Similarly, functions and procedures in the original system become primary candidates for methods and are attached to the aforementioned identified classes. The migration process can be automated to a large extent by using a number of different software analysis techniques. However, no matter how sophisticated the analysis techniques are, user assistance and domain knowledge play an important role in obtaining a viable object model. Moreover, the migration process iterates back to the class extraction phase or class association discovery phase to refine the identified object-

oriented model. Finally, the target object oriented code can be implemented based on the elicited object model.

Global Variable-based Technique: a global variable plays an important role in object mining [Liu90]. For each global variable, the set of routines that directly uses the global variable is determined. After that, a graph is constructed. The nodes of the graph are the routine sets found in the first step, and the edges connect those nodes, which have common references to the same global variables. Each connected component of the graph defines an object and its routines (see Figure 2.7).

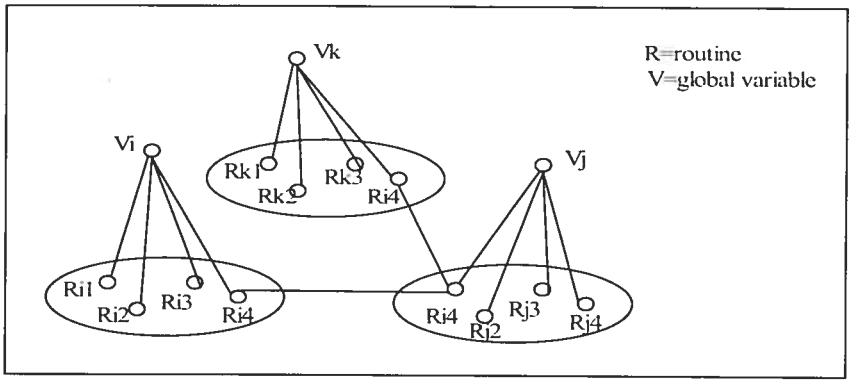


Figure 2.7: Global variable-based object identification

The global type-based technique considers the parameters and return types from procedures [Oga94]. If one of these types is a user-defined type (such as a record type), we may have found an object (corresponding to the type) and its operation (corresponding to the procedure). If a procedure has several user-defined types as its parameters, we prefer larger types to become final objects. A type X is larger than another type Y, if X has Y as its component (such as a field of a record). For example, a type stack may have a type node as its components. We may also prefer such parameter types that are modified inside the procedure [Liv94].

2.4.4 Dominance Analysis

Cimitile et al. proposed a dominance analysis to call graphs to identify candidates for reusable object modules [Cim95]. A node *N* is said to **dominate** another node *M* in a directed graph *G* if each path from the root of *G* to *M* contains *N*. If *N* is a dominator of *M* and every other dominator *N'* of *M* is also a dominator of *N*, then *N* is called an **immediate** or **direct dominator** of *M*. The dominance relationship can be represented as a **dominance tree** where a node's parent is its immediate dominator.

In their approach, cycles (i.e., strongly connected components) are collapsed before applying dominance analysis. It is used to detect additional entities. The algorithm involves the following basic steps:

1. All members of an atomic component are collapsed to a single node (this step is denoted by *Collapse*);
2. Dominance analysis is applied to the collapsed graph;
3. In the dominance tree, each component *C* absorbs its (transitively) dominated subprograms that are not dominated by any other component dominated by *C*.

2.4.5 Discussion

Most of these semi-automatic techniques are based on static analysis of source code. The extracted object model will form the basis for object-oriented re-implementation of the kernel of the system. However, the majority of the research is not concerned about the usability of the derived object model. In fact, the recovered object model includes different types of components. The only reusable part is the business component. How to distinguish it from the rest is still an open question. Moreover, the recovered object model is derived based on a certain criteria. Therefore, the criteria selection is crucial to the final quality.

The existing work all concentrate on one or more aspects of the legacy migration problem. They all failed to provide a comprehensive migration methodology framework to tackle the legacy migration issue as a whole. Some focus on the object identification

techniques, some focus on providing a middle term solution, and some heavily depend on developers who have originally participated in the development of the legacy system. These require us to provide a more suitable solution to solve the legacy migration problem.

Chapter III. Migration Methodology Framework

How can a large legacy software system evolve to benefit from the advantages brought by new emerging technology? Simply rebuilding a new system is unacceptable in various conditions, which has been confirmed by many researches [Ben95][Luc97][Sah99]. With the rebuilding approach, the value that has been invested into the old system will be discarded together with the dispose of a legacy system. This includes the huge amount of former investment that has already been put into the system, and the business logic built inside the system [Sah99][Coy00]. To avoid such loss, an alternative re-engineering approach has been adopted as our research strategy. Our main objective is to efficiently reengineer legacy procedural systems, and make them largely benefit from an advanced modern computing technology: the Object-Oriented technology. Led by this intention, we have developed a systematical re-engineering approach to successfully migrate legacy systems towards object-oriented technology. Software migration is a process of examining and altering a subject system to re-engineer it in a new form [Chi90]. We aim at providing a general-purpose re-engineering framework that improves software quality as a part of the migration process.

3.1 Overview

The goal of our research is to build a disciplined approach to migrate legacy system towards object technology. To be more specific, we aim at developing a reengineering approach in which legacy software can be efficiently analyzed, modeled, and transformed into the object-oriented paradigm.

The research challenges lie in three areas. The first area is the evolution of legacy systems. The second area focuses on how to build computer-aided techniques to facilitate the migration. Finally, the third research challenge is how to construct a comprehensive migration supporting system, thus to greatly support the collaborative team work of the migration process. The migration methodology framework is illustrated in Figure 3.1.

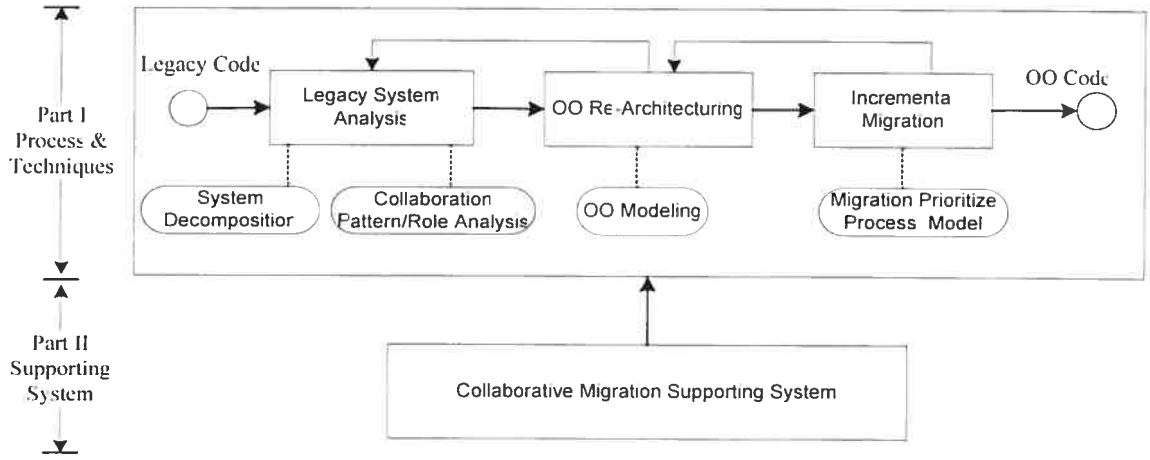


Figure 3.1: Migration Methodology Overview

Our approach is composed of two major parts: the first part is the progressive migration process and techniques, which are encompassed in the upper rectangle in Figure 3.1; the second part is the collaborative migration support environment, which is indicated in the lower rectangle. Each part has its own concentration on different aspects of migration activities. The first part deals with the migration process and techniques; the latter one provides an integrated migration project supporting system to manage cooperation and management issues. These two parts organically form a comprehensive solution for practitioners to efficiently conquer the legacy system migration problem.

3.2 Ideal Migration Process Model

During the whole migration course, at different time periods, the tasks are different. We have designed an ideal migration progress model to map these differences. Our ideal

model is used to facilitate the understanding of detailed migration procedures that might be introduced in any legacy migration project, in which the migration condition is supposed to be ideal. This means that, the project resource are sufficient, such as there is no shortage of time, human, budget, etc. The ideal migration process model is composed of two major parts, namely a reverse engineering part and a forward engineering part. They are illustrated in Figure 3.2 and Figure 3.3.

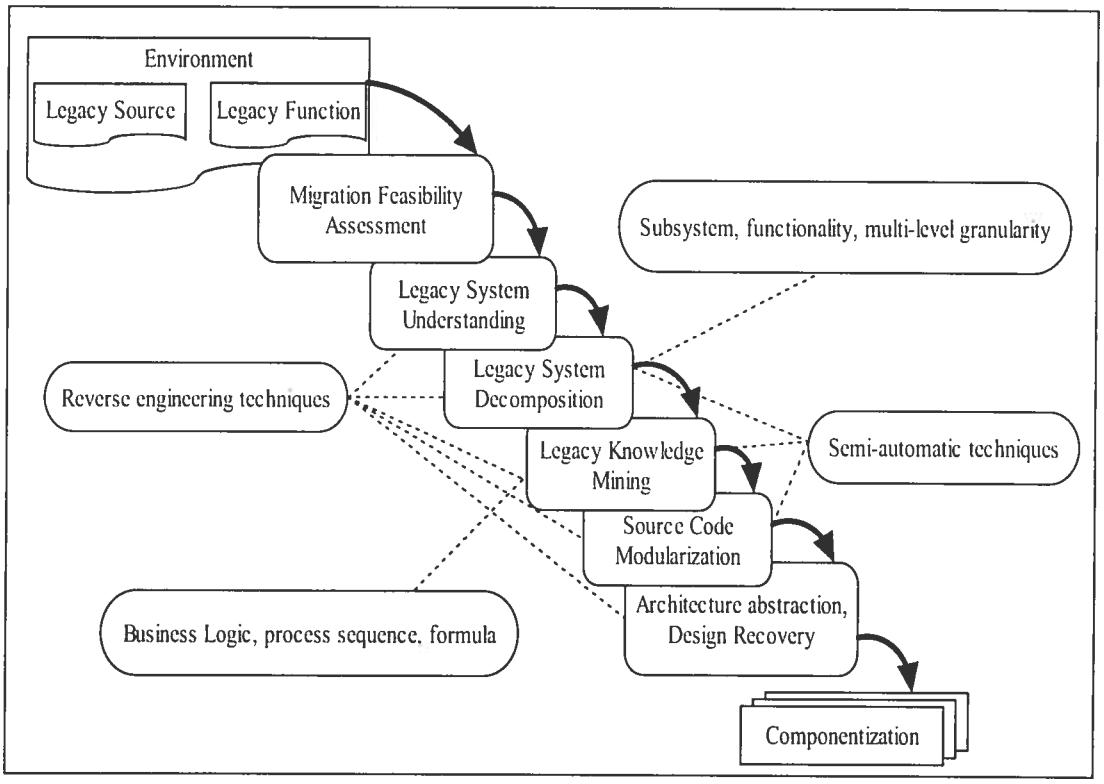


Figure 3.2: Ideal Migration Process Model - Reverse Engineering.

The reverse engineering part (shown in Figure 3.2), starts at the migration feasibility assessment. In this stage, the migration risks and feasibility are analyzed to evaluate the benefits of the migration project. The following evaluations are performed:

- Assessment of the cost-benefit of the migration project;
- Estimation of the risk factors: resources, migration plan, human, technology, inner and outer environments, disaster event prediction, etc.;

- Estimation of possible migration strategy failures: migration requirements, environment integration, migration process, legacy knowledge mining.

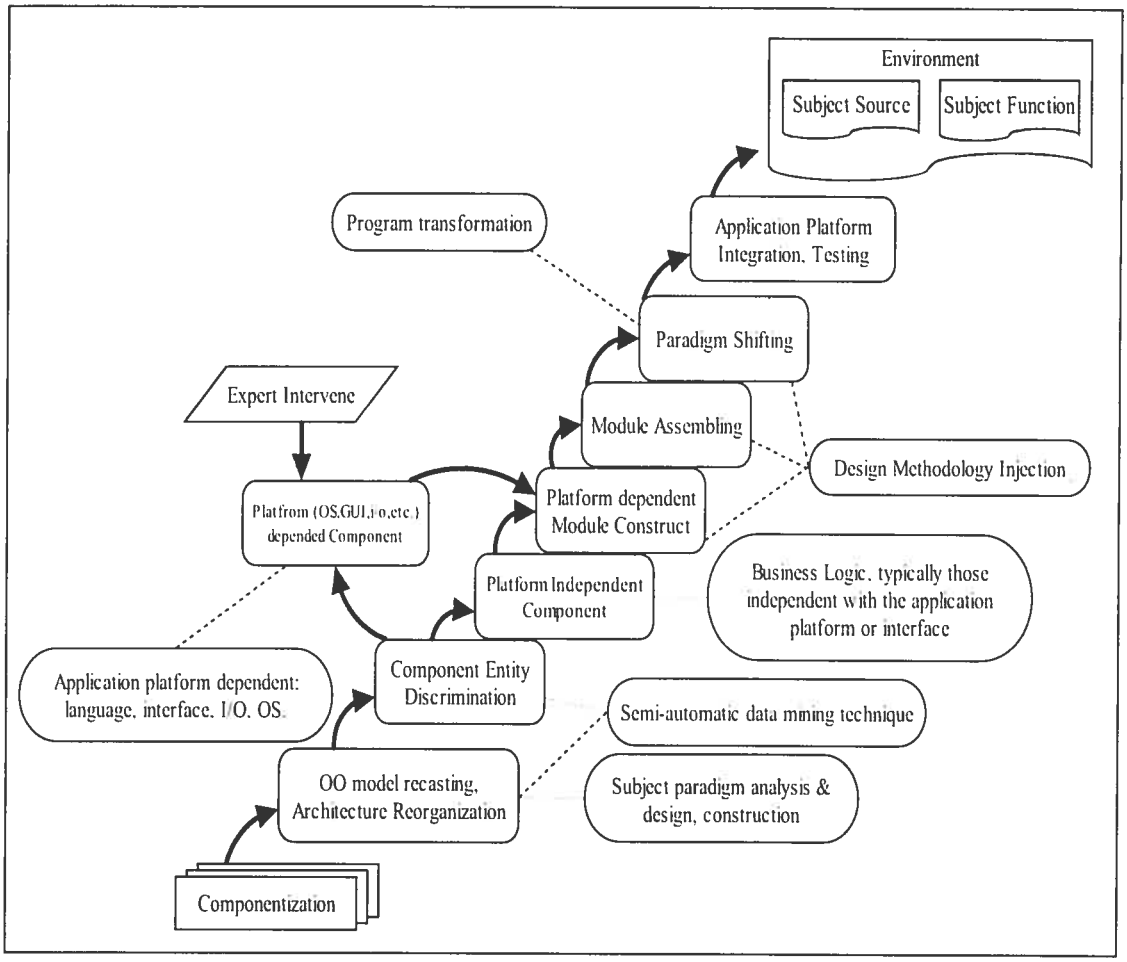


Figure 3.3: Ideal Migration Progress Model - Forward Engineering Part

The next process is the legacy software analysis, which is further divided into three detailed stages, namely legacy system understanding, legacy system decomposition, and legacy knowledge mining. We have developed dynamic analysis tools to facilitate the legacy system analysis process. After these stages, the source code modularization will distill the coarse-grained object-oriented model from the legacy source code based on the results from former three stages. The recovered legacy architecture abstraction and design elements will be revised in an object-oriented perspective according to the decomposing

and the distilled object model of potential class candidates. The reverse engineering result is stored into a repository.

In the forward engineering part (see Figure 3.3), a comprehensive object-oriented modeling and architecture reorganizing step is performed to establish the object-oriented design, which is based on the recovered artifacts from the former reverse engineering. Semi-automatic techniques will be used to facilitate this stage. Since legacy systems are always executed in particular environments, there exist two kinds of components, one is platform dependent, and one is platform independent. For the former one, extensive expert intervention is necessary to carry out those specific modeling tasks related to different application environments; for the latter one, we have developed three kinds of techniques to facilitate the final refinement of object-oriented model. These techniques will be detailed in Chapter 5. In the model assembling stage, both platform dependent and independent components are integrated into an assembled comprehensive model. Finally, the object-oriented system implementation is achieved by the paradigm shift stage. An incremental implementation approach is applied at this stage. Within the following 3 stages, namely object-oriented design refinement, object model assembling, paradigm shifting and implementation, the design methodology injection process will embed the enhanced object-oriented design artifacts into the final object model, thus to improve the quality of the target migration system.

3.3 Practical Migration Process Model

The practical migration process model is developed based on the ideal migration process model. It is designed to facilitate the normal migration practices which usually have many restrictions on the resources. This model eliminates those small but detail steps in ideal model, and modifies some steps to facilitate the usage of the techniques and tools that we have developed. It combines the reality issues with the technical issues that we applied in our research. It is illustrated in Figure 3.4.

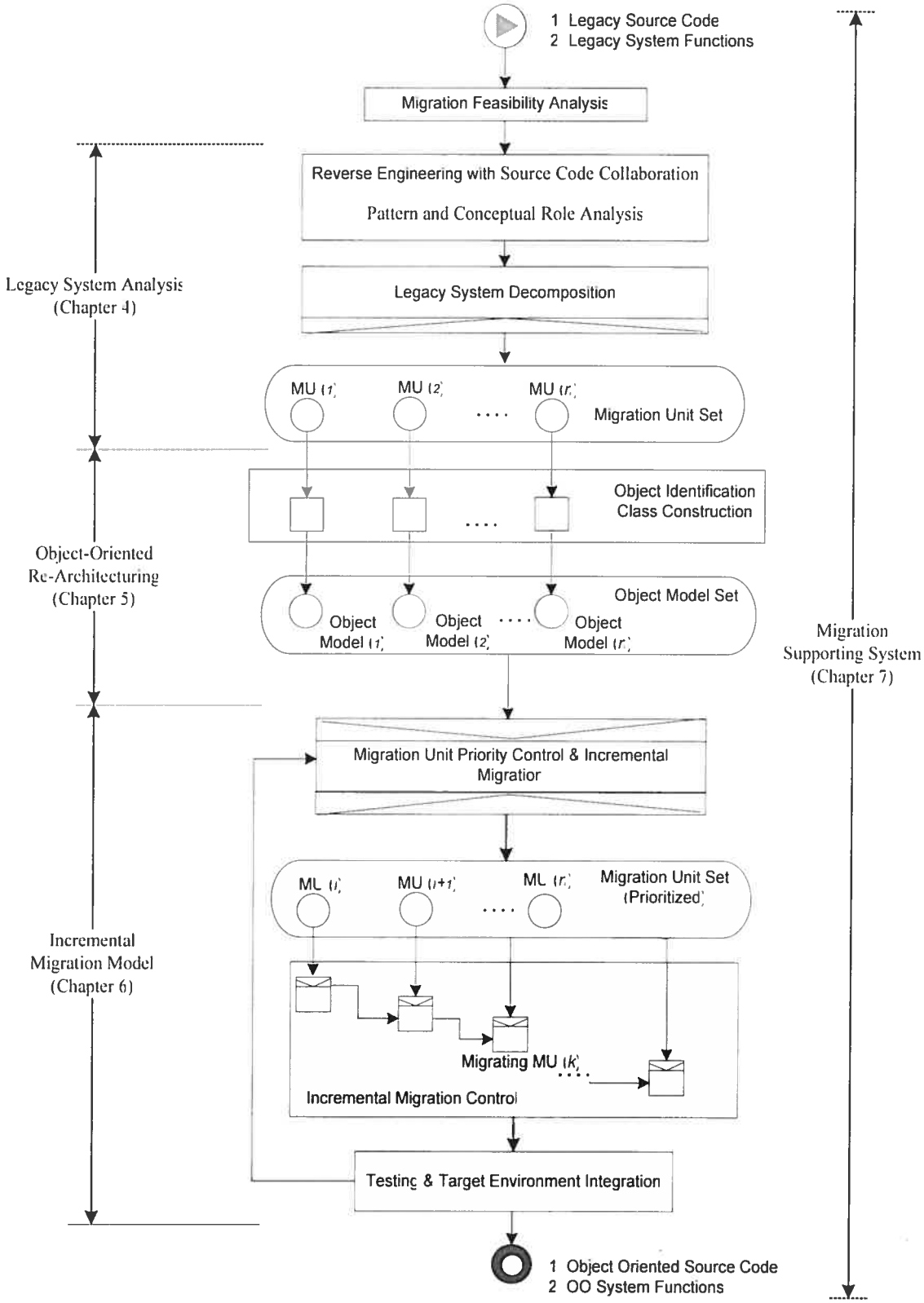


Figure 3.4: Practical Migration Process Model.

The starting point is the legacy system with its source code and legacy functions. The end point is the transformed object-oriented source code and OO system functions. Before the start of all the other process actions, a comprehensive migration feasibility analysis will be performed to evaluate the risks and cost-benefits of the migration project. The practical migration process model contains the following four modules:

1. Legacy System Analysis

In this module, we focus on two parts. One is the legacy system decomposition, and the other one is legacy source code collaboration pattern and role analysis. The system decomposition focuses on how to decompose the legacy system into parts, thus facilitating the next stage of applying a divide-and-conquer approach to implement the legacy system re-architecturing and incremental migration. The decomposition strategies are constructed based on different emphases of system analysis aspects. The code collaboration pattern and role recovery concentrates on the identification of legacy source code features and constructional structures. Each recovered collaboration pattern represents a concrete implementation block of the observed system functionality. The recovery of roles further supports the decomposition of the whole system into a role-based hierarchical representation.

2. Object-Oriented Re-Architecturing

In this module, the focus is on the object-oriented modeling of each migration unit (decomposition part) of the system. To solve the object identification problem, we have developed three major techniques, namely the rule-based class recovery, static featuring technique, and dynamic featuring technique respectively. For static featuring techniques, we have applied two specific object identification algorithms which are based on the genetic approach and the conceptual clustering algorithm. In legacy system object-oriented re-architecturing practices, we should apply all of these techniques to solicit the object models, and compare, analyze, evaluate the results with human intervention, and eventually optimize those results to obtain an optimum solution.

3. Incremental Migration Model

According to the decomposition strategies, the whole system is broken into logically cohesive partitions. Various system decomposition outcomes will further be analyzed, compared and optimized with human intervention, and synthesized to generate an optimum solution. The final system partition result will eventually produce a set of migration units (MUs). The incremental migration process is designed to contain multiple iterations, each of which will only focus on migrating a certain part of the target system (in our case, the MU), leading to an increase in the portion of the renovated code and to a respective decrease of the legacy code. In this model, a fuzzy expert system is applied to prioritize the migration sequence.

4. Migration Supporting System

Web-based collaborative migration project involves many people working together without the barrier of time and space differences. However, the large scale of collaboration in a typical migration project lacks of sufficient supporting techniques to facilitate project planning, monitoring distributive collaboration tasks and communication. The prototype of migration supporting system, *Caribou*, is designed to tackle these three important issues.

3.4 Integrated Migration Support Environment

In a large legacy migration project, a sophisticate migration support system is keenly needed. We provide a migration support environment, called *Caribou*, to systematically support migration project planning, task monitoring, and team collaboration. Meanwhile, it also works as an integrated environment to accommodate various tools and techniques applied in the detailed migration tasks. The ultimate objective of developing such a system is to enhance the efficiency of management, and maximize the possibility of success for any migration project. It also provides migration tools support, automatically collects and analyzes a range of process/project task data, accommodates quantitative migration control, etc. It provides the following important functionalities to facilitate collaborative teamwork:

- *Migration task definition:* The monolithic legacy source code will be divided into MUs (*Migration Unit), W*fMS will manage the process for each task.
- *Management of migration supporting tools*
- *Task control for individual team member*: In our migration support system, workflow engine will read workflow definition, execute, and keep the track of the work. The tasks assigned to an individual programmer in a team will be well organized.
- *Request/response control between team members:* The individual maintainer's question will be processed through internet. The interaction of team member's requests and responses will be conducted through Caribou's collaborative communication control module. This will reduce the workload of request enforcement between team members, and will facilitate the problem tracing ability.
- *Administration and monitoring of the whole project progression:* It allows the migration project manager to trace the status of each migration unit (MU), migration task, and the whole task workflow. With the progression of the migration project, historical data will be daily collected, analyzed, and saved. The dynamic changes and analysis results will be generated to facilitate the control of teamwork. When any calculation result exceeds the threshold, alarming information will be sent to the related persons, the critical part will be pinpointed and causation analysis will be provided by automatic tracing methods to find out the possible origin.

3.5 Summary

In this chapter, we have presented the blueprint of our migration methodology framework. It is constituted with four major components, namely legacy system analysis, Object-Oriented re-architecturing, incremental migration model, and migration supporting system. The details for these four components will be presented in the following four chapters.

Chapter IV. Legacy System Analysis

When we are facing a legacy system migration project, maintainers inevitably encounter the difficult task to understand legacy software system. Hall's research shows that understanding the documentation and logic of programs occupies about 50 to 60 percent of the maintenance programmers' time [Hal88]. In many cases, even the original developers find it hard to understand their own code after a long period of time [Som00]. As a consequence, migration tasks tend to be difficult, expensive, and error prone [Mic03].

Therefore, the analysis of legacy system becomes a very crucial part in a migration project. In our research, we focus on following two aspects of legacy system analysis: (i) legacy system source code collaboration pattern, and role recovery and analysis; (ii) legacy system decomposition. The code collaboration pattern and role recovery concentrates on the identification of legacy source code features and constructional structures. This work facilitates the comprehension of how legacy functionalities are implemented by means of source code collaboration, and the patterns/roles of various types of code cooperation forms. The system decomposition makes it possible to further apply our divide-and-conquer approach to implement the legacy system Object-Oriented re-architecturing and incremental migration.

4.1 Source Code Collaboration Pattern and Role Analysis

Software functionalities and behavior are accomplished by the cooperation of code artifacts. The understanding of this type of source code collaboration provides an important aid to the maintenance and evolution of legacy systems. However, the original collaboration design information is dispersed at the implementation level. The extraction

of code collaborations and their roles is therefore an important support in legacy software comprehension and design recovery. In this section, we present our approach to recover and analyze code collaborations and roles based on dynamic program analysis technique. We will also present the experiment using the tools that we have developed to carry out our approach in Chapter 8: Experiment and Evaluation.

4.1.1 Approach Introduction

Large legacy procedural systems are normally organized in a structured form [Let99][Lak97]. Code is divided into separate source files based on different design criteria [Pau94][Let99]. For example, in Cobol, Fortran, C, and Ada, the functions that relate to the same topic (such as "error") are usually grouped together into a single program file. Source files are further structured into different directories according to the functionalities they contribute to [Let99]. This kind of program code organization reflects the original legacy design rational [Pau97]. Each source file and directory represents a certain design concept [Let99]. Each code cooperation instance contains a limited number of such code units. We view these construction units as source code modules which interact with each other to realize the system functional behavior [Let99]. Moreover, each module plays a set of conceptual roles inside of the cooperation. The role relationship among code modules reflects the control characters of source code; and the code collaboration pattern reveals code organizational structure.

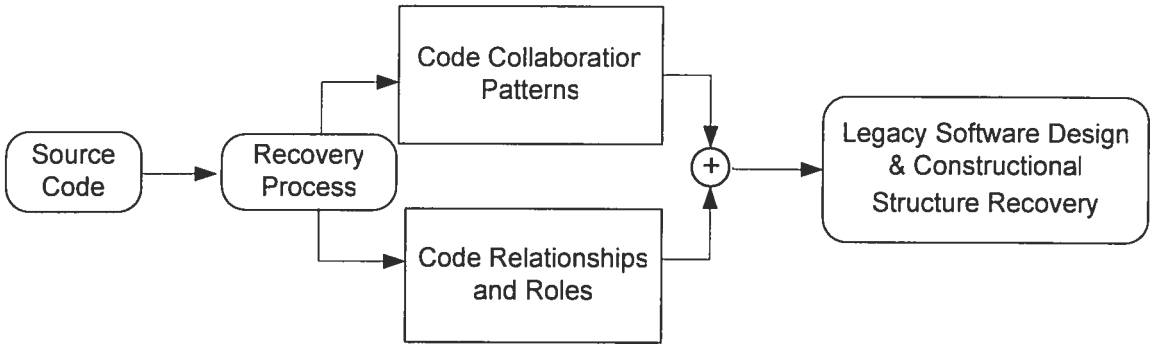


Figure 4.1: Legacy System Design and Constructional Structure Recovery

Recovering such code collaboration patterns and conceptual roles from code artifacts is hereby an important aspect for better understanding and re-engineering legacy code [Ric02]. It further facilitates the recovery of legacy software design and constructional architecture, as illustrated in Figure 4.1. However, the large number of code modules and the complexity of dynamic relationships make discovering and analyzing module collaboration patterns and code roles a difficult task.

4.1.2 General Concepts

In this section, we introduce the underlying analysis concepts, terminologies, and formalisms that we have used in our research on legacy system analysis.

Source code module: The source code of a system is usually organized in a structured form [Let99][Lak97]. Code units related to the same concept or topic lay in a single source file and are further stored into different directories, which reflect the original design rational [Let99]. We view source file or directory as source module, or simply module.

Interaction instance: An interaction instance is a dynamic information transaction between two modules. It triggers a message flow from sender module to receiver module.

Collaboration instance: A collaboration instance is the sequence of contiguous interaction instances, which together form a chain of events.

Collaboration (or cooperation) pattern: A collaboration pattern is a frequently repeated series of several collaboration instances. During the whole process of interactions, modules show strong cooperative forms: certain modules always cooperate together to implement a particular type of task. We view this kind of phenomenon as a collaboration pattern (see the detailed analysis terminology in Figure 4.2).

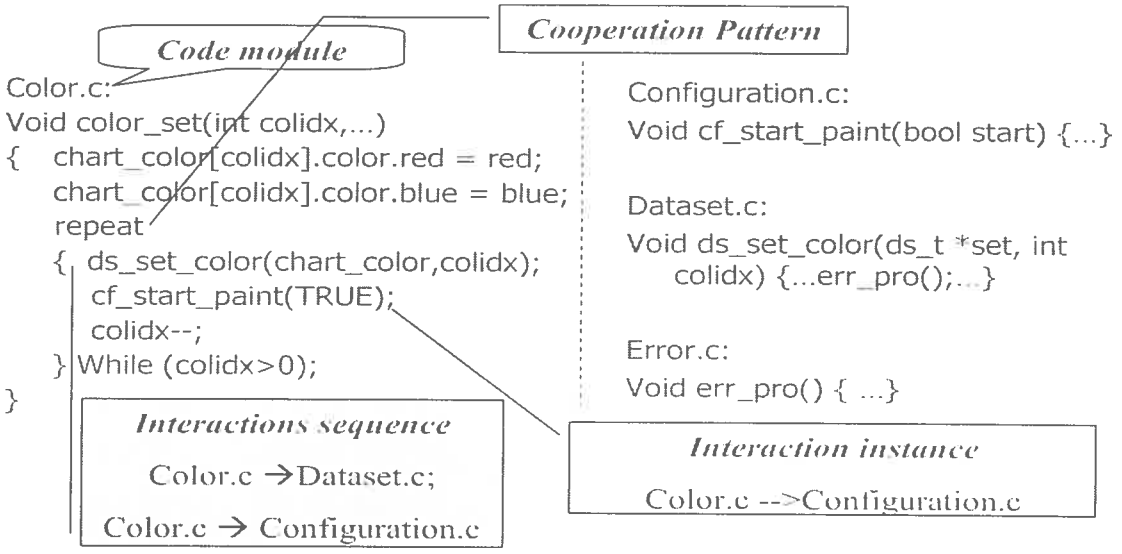


Figure 4.2: Illustration of Analysis Terminology

Dynamic trace record: The dynamic trace record is the program trace information that we have captured during the execution of the target legacy system. A sample segment of the trace is given in Table 4.1.

Fun_id,	Level,	Module,	Routine,	Direction
3,	5,	indateentry.c,	in_dateentry_set_text,	In
3,	6,	intransinp.c,	on_input_data_changed,	In
3,	7,	transaction.c,	*trans_get_typelist,	In
3,	7,	transaction.c,	*trans_get_typelist,	Out
3,	6,	intransinp.c,	on_input_data_changed,	Out
3,	5,	indateentry.c,	in_dateentry_set_text,	Out

Table 4.1: Dynamic Tracing Data Format

An interaction instance includes six major components, namely system functionality id, invocation (call) level, sender module, receiver module, invocated routine and direction. In Table 4.1, the "Module" represents the receiver module at that call level. The sender module is the module that locates at the nearest former interaction instance which has one call level less than the call level that current interaction instance has. The Fun_id stands for system functionality identification number, which correspond to the

specific system functionality that was performed. The level represents the invocation depth. Direction is the orientation of message flow. The module and routine represent the module name and the function/procedure that carries out the invocation event.

Conceptual role: A conceptual role is the predictable stereotype of an individual module. It represents the general characteristic of the module's utility in the program.

Role definition: The definition of a specific role that a code module plays in a collaboration pattern reflects the relationship between two modules. A particular module may have multiple roles in different relationships with other modules, but normally it has a major dominant role. From the construction point of view, a simple job-dispatch relationship can be metaphorized into "manager-worker" roles for the sender and receiver modules. This role pair relation explains that the module with a high level "Manager" role dispatches tasks to the modules with a lower level "Worker" role. We mainly define four pairs of conceptual role relationships based on the invocation contributions among the relationship.

4.1.3 Collaboration Pattern and Role Recovery Approach

In this section, we present our program analysis approach for the recovery of code collaboration patterns and conceptual roles from source code artifacts. We apply dynamic program analysis and software visualization techniques to accomplish our goal. We have developed two reverse engineering program analysis tools, namely *Dynamic-Analyzer* and *Collaboration-Investigator*, to carry out our approach. The tools are used to automate the process of detecting, recovering, and analyzing legacy source code collaboration patterns and roles.

Collaboration and conceptual role are two design concepts that have been scattered throughout source code [Ric02]. Inside collaborations, participant modules interact with each other to carry out specific tasks. The cooperation is confined in an interaction structure form, which describes a set of allowed collaboration behaviors for each module.

Such structure is implemented with two major design concepts and dispersed in code: the repetitive code cooperation pattern and conceptual role. Each participant plays a certain type of role in the collaborations. Recovering such information can largely facilitate the program comprehension process and promote program analysis into a deeper level. Since procedural languages do not provide explicit means to capture such design information, maintainers have to heavily rely on human efforts to investigate these design logics in legacy software.

To recover collaborations and roles from legacy source code, we propose an approach that uses dynamic analysis, software visualization, and automatic/semi-automatic detection techniques to achieve our goal. It is illustrated in Figure 4.3. We first execute system functionalities separately, and capture dynamic interaction information among modules during period of system execution.

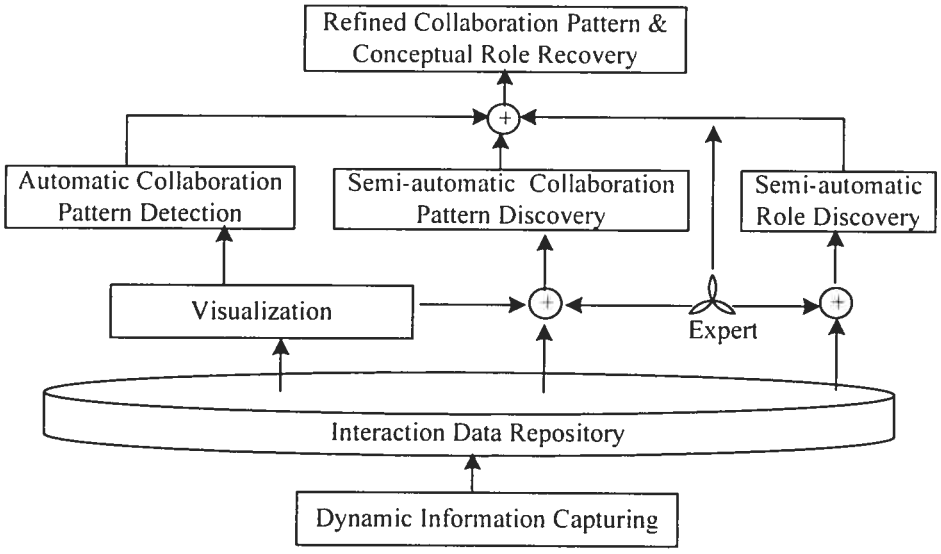


Figure 4.3: Recovery Approach Schema

Then we apply software visualization techniques to analyze and identify dynamic program features. Later on, automatic pattern detection process is performed to recover all significant repetitive collaboration instances. Meanwhile, with the intervention of maintainers, the semi-automatic process detects the collaboration pattern and participants' roles, and investigates their features. In addition, a cross-check and

refinement process is conducted to combine the two outcomes, and distill the final refined results. The following are the key issues addressed in our approach.

Dynamic information capturing. We apply a dynamic analysis technique [Bal99] to capture module interaction messages, data transformation routes, and control flow information during program execution.

Software visualization. We apply graphic simulation to represent the captured information into a more understandable visual form as a set of comprehensive graphical diagram views. There are two kinds of information we visualize: the first one is the pure interaction information that represents the behavior of the code; the second one is statistic data information. For the first one, we use both static visualization and animation to simulate the dynamic nature of code artifacts cooperation. For the latter one, we use graphical diagrams and graphs to visualize the statistic analysis results.

Automatic and semi-automatic collaboration pattern and role detection mechanism. With the results from the former two processes, we are able to study the features of dynamic code interaction instances, such as the components of the code cooperation, their directions, the serials of code collaboration sequence and their frequencies, etc. To discover the collaborations and conceptual roles dispersed over the huge amount of code transactions, we have to adopt some strategy to limit the searching space. The difficulty lies in the efficient identification of those significant repetitive interactions, which jointly form a meaningful collaboration pattern and role relationships in the large transaction space. We apply our automated detection technique on the visualization results, and extract fine-grained collaboration patterns. The advantage is that it is capable to detect a wide range of collaboration patterns, while the disadvantage is that the maintainer may lose the control of expressing her emphasis on discovering patterns. As a remedy to this shortage, we adopt a semi-automated recovery of collaboration patterns and roles with human intervention.

According to the maintainer's emphasis, she can interactively select the recovery criteria, which gives the preferable weight on different aspects of automated pattern

recovery. The collaboration pattern and role relationship discovery results will reflect the emphasized interests of the maintainer, thus only recover those collaboration patterns and roles which exhibit the most interesting features specified by human. Finally, these two types of outcome will be compared and refined to produce the final recovery result.

In order to efficiently recover collaboration patterns from execution trace information, we have to design certain types of criteria to emphasize what aspect is more important in detecting which sequences of collaboration instances are related, and thus may be further combined together to form a concrete collaboration pattern. The criteria are divided into the following three categories:

Interaction instance component. Recall that an interaction instance includes six major components, namely system functionality id, invocation level, sender module, receiver module, invocated routine, and direction. Based on different emphasis, the maintainer may use any combination of these components to define the recovery criteria.

Collaboration instance selection. The main purpose of finding collaboration instance is to recover the invocation path. For this reason, we may select to view only the interaction instances involving specific modules. Meanwhile, to limit the observation scope, we may also define a consideration boundary that confines the recovery process within a certain depth range.

Collaboration pattern matching. When several frequently repeated collaboration instances form one collaboration pattern, the shape of that pattern may not be unique. Different interaction sequences, may lead to various visual outlines, while the semantics of these patterns are identical. Therefore, we define the criteria to let our tools to compare two collaboration patterns.

4.1.4 Automation with the *Dynamic-Analyzer*

As we discussed in the introduction, static analysis does not present sufficient

information to study the interactions of source modules. Recording dynamic information of a program can provide us with sufficient knowledge about message exchanges during the program execution period. However, this technique faces two major issues: the overwhelming volume of tracing data and incomplete coverage of the code. In our approach, since the focus is on a limited set of system functionalities and behaviors, the dynamic coverage contains only the relevant code artifacts. In fact, this turns out to benefit the resolution of the first issue to reduce the volume of tracing data. Based on the approach we presented in the former section, we have developed a reverse engineering tool, the *Dynamic-Analyzer*, to automate the dynamic capturing and visualization of dynamic source code message process information among source modules (see Figure 4.4). First, the legacy source code is instrumented to record execution information. Then, the interesting system functionalities are executed to observe system behaviors; meanwhile, program dynamic information is retrieved, processed (normalized), and fed into the data repository.

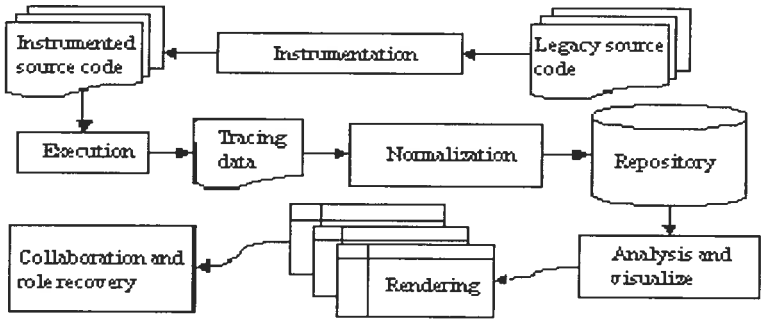


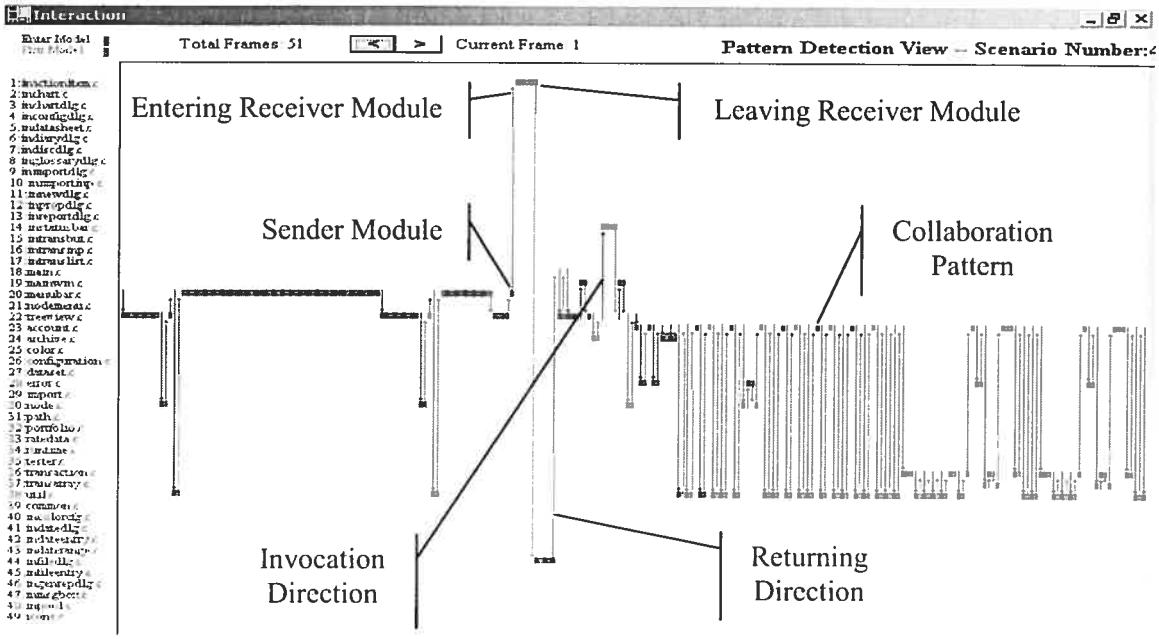
Figure 4.4: Workflow of the Dynamic-Analyzer

Later on, the visualization and animation program will present the information through visual effects to provide a meaningful way to investigate the interactions. Finally, the automatic collaboration pattern detection process will be performed to distill all the patterns. The *Dynamic-Analyzer* can automatically discover all fine-grained collaboration patterns. We have also developed another reverse engineering tool to incorporate human intervention in the discovery process, and combine the outcomes from the *Dynamic-Analyzer* to improve the result. That semi-automation approach will be detailed in

following section. One desired feature of the *Dynamic-Analyzer* is both observed system and analysis tool will run in parallel. Maintainers are now able to observe system behavior and the visualization of source module interactions/patterns at the same time. In this way, we can directly relate any specific system behavior to the visual effects of module interactions in a real-time manner, thus reduce the mental efforts to remember and match these two subjects.

We use the *Dynamic-Analyzer* to define different types of views to exhibit information at different granularity levels, and to facilitate the smooth navigation among those levels. We mainly visualize two types of information: the pure interactions, and the statistic data information. For the first one, we apply static and animated visual effects to enhance the recovery process. Figure 4.5 illustrates the fine-grained dynamic module interaction footprint view and animated pattern detection views produced by the *Dynamic-Analyzer*. The left-most vertical part shows the name of modules; the horizontal direction represents the time sequence; the dark (red) box indicates an invocation interaction instance from the sender module; the gray (green) box shows the return of interaction instance from the receiver module; the dark (red) line with direction point shows an outgoing message from the sender module towards the receiver module; the gray (green) line with direction point represents the returning of the interaction message from the receiver module back to the sender module.

(i) Dynamic Module Interaction Visualization: Footprint View



(ii) Automated Collaboration Pattern Detection

Figure 4.5: Footprint (i) and pattern detection (ii) views from Dynamic-Analyzer

We employ the *Dynamic-Analyzer* to automatically detect all the repetitive serial of collaboration instances, and distill them as candidate collaboration patterns. To further reveal the construction structure of particular system functionality, we need more effective means to discover the dynamic module interaction space. Our approach is to visualize the dynamic information in a form that illustrates the relationships between difference source code modules which are involved in the activities that generate the specific system functionality.

As demonstrated in Figure 4.6, the visual representation of the comprehensive module interaction relationships reveals a system constructional structure that implements the observed system functionality.

- **The invocation level:** corresponds to the call depth from sender module to receiver module.
- **The link between modules:** represents the invocation instance from higher level module to lower level module.

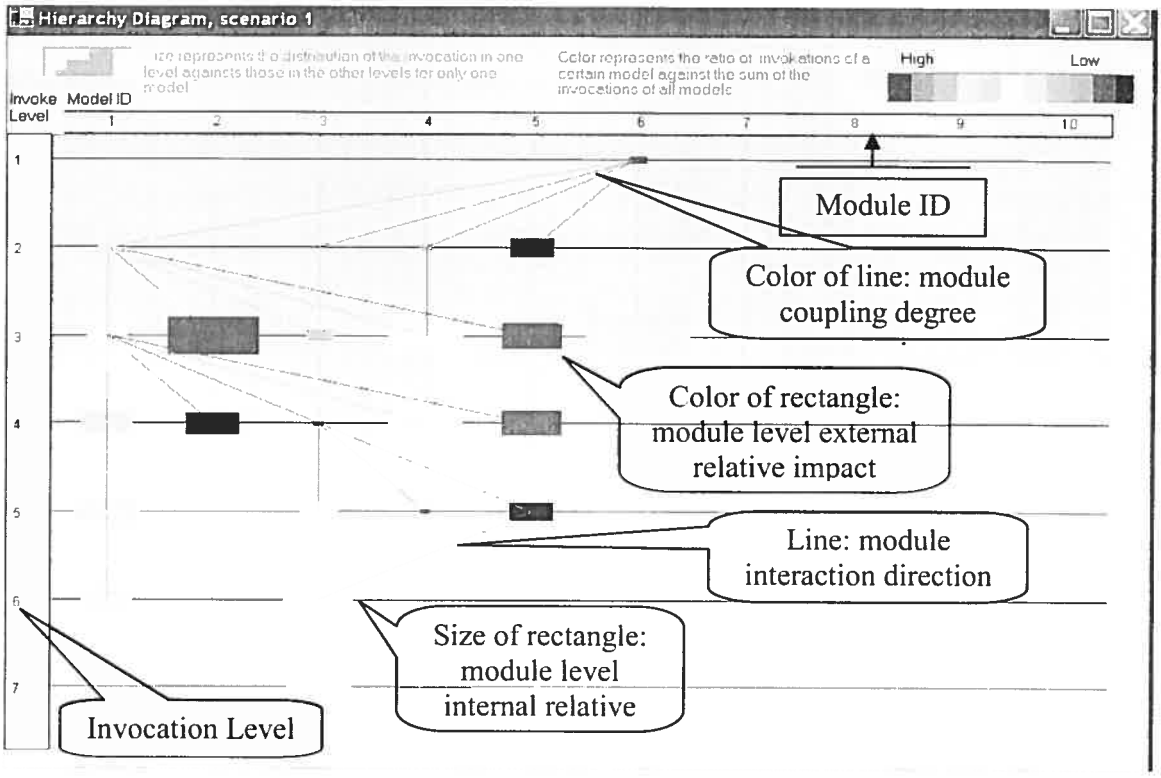


Figure 4.6: Construction Structure View of System Functionality

- **The location of rectangle:** shows at the specific location (invocation level and module), there is a certain amount of module invocation activities.
- **The size of rectangle:** the rectangle area stands for the percentage of invocations the module has at the particular level, compared with the total number of invocations that module has among all the levels. Suppose the "Full_Size" rectangle has 1cmX1cm height and width. The mathematic formula is expressed as following:

$$Size(Module\_a, Level\_b) = Full\_Size * Invocations(Module\_a, Level\_b) / Invocations(Module\_a, All\ Levels) \quad (4.1)$$

Here, the Full_Size represents the maximum rectangle space between two nodes. The size of each rectangle shows the relative impact of invocations that a module has at certain level, which indicates the activity intensity degree at each invocation depth for one particular module.

- **The color of rectangle:** reveals the module at certain level external relative impact, which specifies the activity intensity degree ("weight") at each invocation level for one particular module in comparison to the total invocations of all modules. Color scale schema is applied to reflect the weight. The notation of color is shown in Figure 4.7.
- **The color of link:** illustrates the coupling degree of these two linked modules, which reflects the weight of that link. For the color of link and color of rectangle, we apply 10 color scale schema to symbolize the weight variance gradient from "High" to "Low". Figure 4.7 illustrates the color representation scheme of the weight variance gradient.

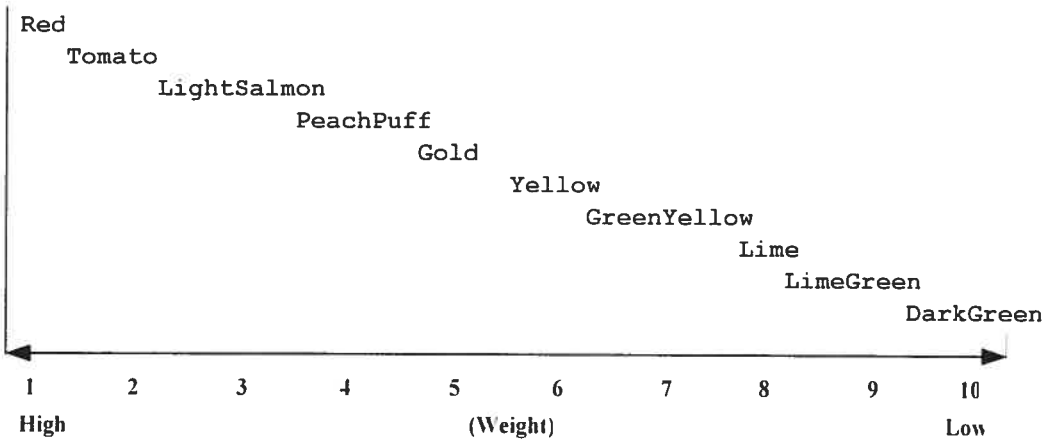


Figure 4.7: Color Representation Scheme of Weight Variance Gradient

The color of both link and rectangle uses "weight" to represent the percentage of invocations it occupies compared with the total number of invocations that all modules have. The mathematic formulas are expressed as following:

Weight_Link(a,b)= 10 * Invocations (Module_a →Module_b) / Invocations (All Modules)	(i)
Weight_Rectangle(Module_a, Level_b) = 10 * Invocations (Level_b) / Invocations(All Modules)	(ii)

(4.2)

4.1.5 Recording System Functionality Scenario

When we analyze system functionalities and their source code construction structure, it is desirable to record system behavior for the comparison and analysis. Therefore, we provide a scenario recording function for *Dynamic-Analyzer* to capture the screen snapshots of system functionality and behavior (see Figure 4.8).

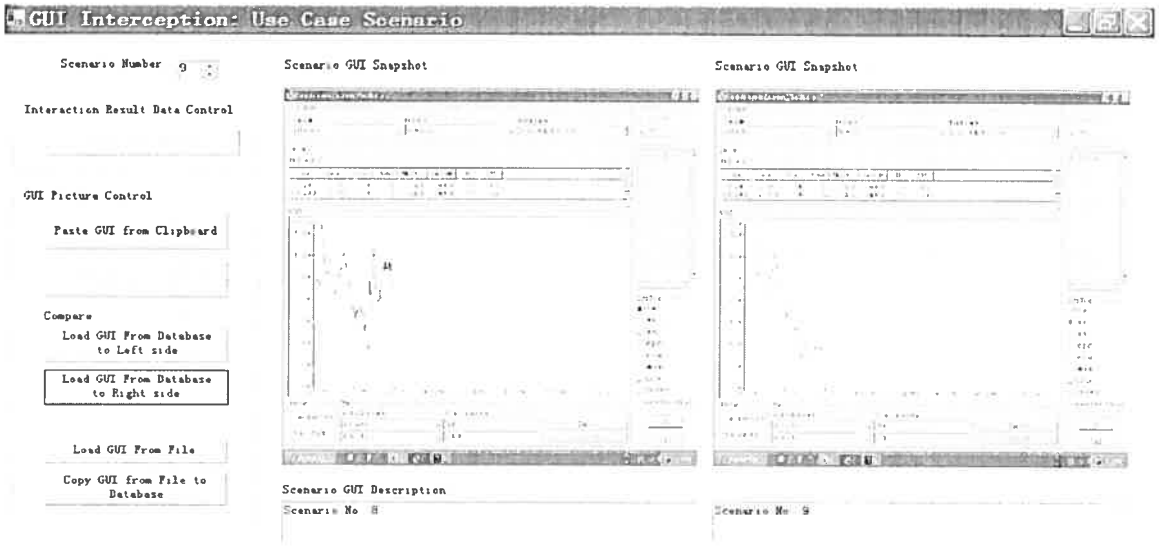


Figure 4.8: System Functionality Scenario Recorder

The recorded scenario screen snapshots will be labeled and stored into the repository database for analysis usage. Later, when we perform the dynamic visual analysis, we are able to retrieve the scenario pictures to reflect the observed system functionality, thus to link the system behavior and the visual analysis results (views and diagrams). In this way, we do not need to execute the target system every time when we want to study it.

4.1.6 Analysis with *Collaboration-Investigator*

As we have discussed in section 4.1.2, a collaboration pattern is a frequently repeated serial of collaboration instances, which involves different modules cooperating together

to perform a certain type of work which contributes to the implementation of observed system functionality. During the whole process of interactions, those modules involved in the pattern show strong collaboration manners: they cooperate together to implement a particular task. To study coloration patterns, and further analysis the roles of participant modules, we have designed another reverse engineering tool *Collaboration-Investigator* (see Figure 4. 9) to carry out our approach explained in section 4.1.3.

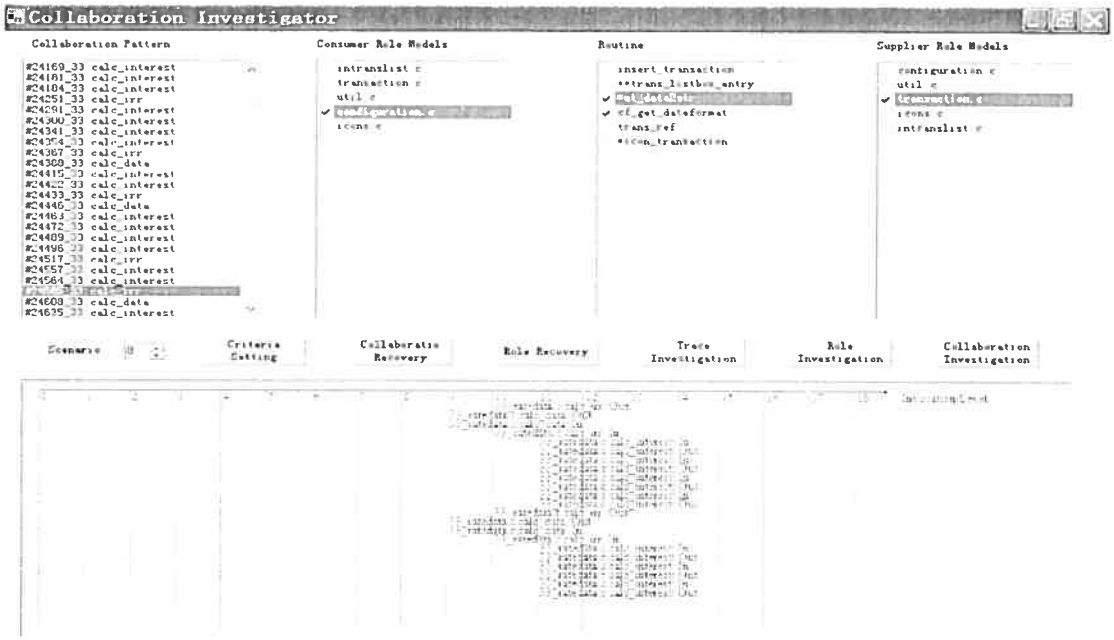


Figure 4.9: Collaboration-Investigator: a Reverse Engineering Tool for Collaboration Pattern and Role Recovery and Analysis

Seven major tasks are carried out for the recovery and analysis of collaboration patterns and conceptual roles. They are:

1. Pattern criteria establishment,
2. Interaction investigation,
3. Collaboration pattern recovery,
4. Visualization of collaboration pattern,
5. Role recovery,
6. Collaboration pattern investigation,
7. Role investigation.

Collaboration Pattern Criteria Establishment: We have set up three categories of pattern recovery criteria, namely interaction component selection, collaboration instance selection, and pattern matching. The choice of optional items in each category reflects the maintainer's observation emphasis of pattern recovery aspects. The naming convention of distilled collaboration pattern is the unique sequential id number plus the first sender module's name and the first invocation routine name.

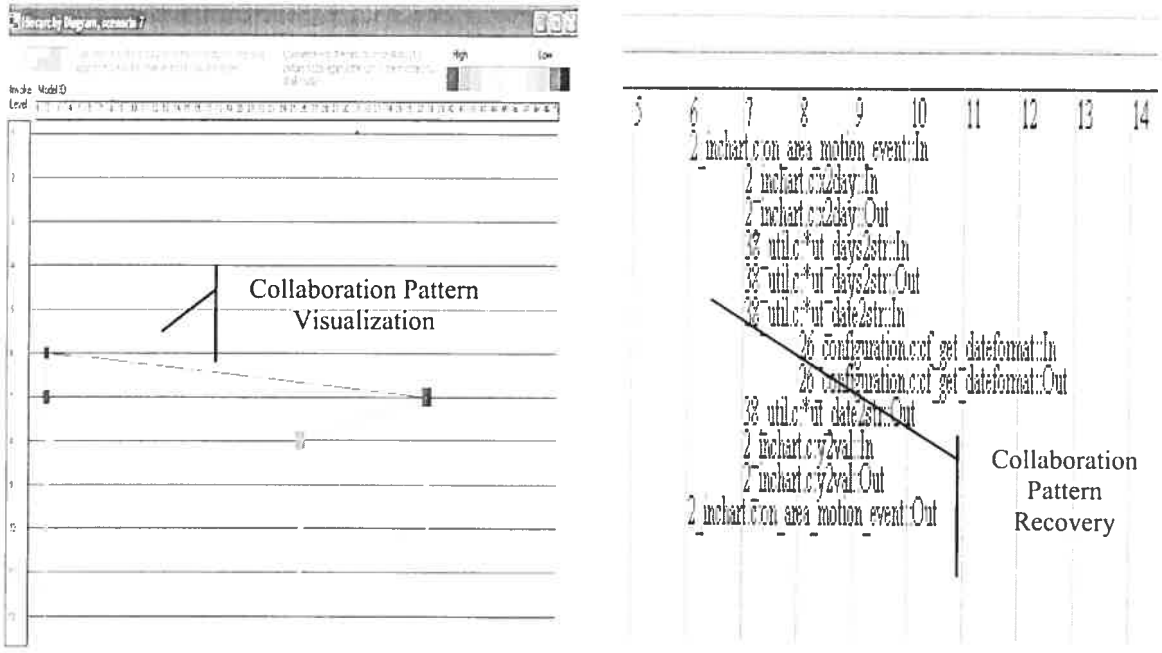


Figure 4.10: Collaboration Pattern Recovery and Visualization

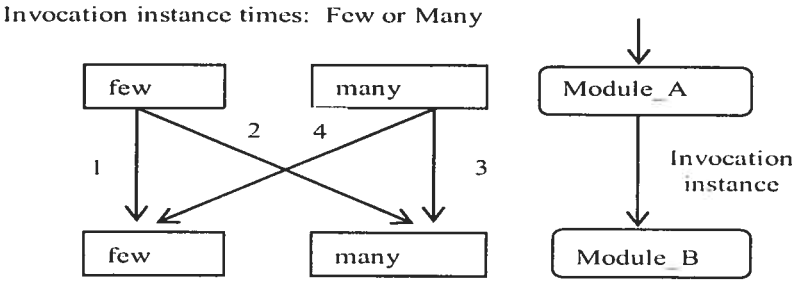
Interaction Investigation. It is designed to explore all the components of any interaction instance. When we select the pattern name, the sender module and the receiver module, we can generate all the routines (functions and procedures) that are invoked from the sender module to the receiver module within the selected collaboration pattern. If we only pick up the routine name, it will also be able to generate the full interaction information in the collaboration pattern which contains that routine. Through this component, we will be able to fully explore the module interaction space.

Collaboration Pattern Recovery: This component recovers the collaboration patterns based on the criteria that have been previously created.

It identifies those significant repetitive interactions, which jointly form a collaboration pattern and role relationships in the large transaction space. The result will be shown in the "collaboration pattern" frame (see Figure 4.10).

Visualization of Collaboration Pattern. The *Dynamic-Analyzer* is used to generate the visual effects of recovered collaboration pattern. It also generates the visual representations of the corresponding collaboration instances for a pattern. A sample recovered collaboration pattern and its visualization is shown in Figure 4.10.

Role Recovery: Based on the recovered collaboration patterns, the module relationships and the module roles will be defined according to the role pairs classification illustrated in Figure 4.11. For each pattern, the participants' module role table will be generated. It recovers four role relationship pairs based on module invocation frequency: "director-manager" (few, few); "manager-worker" (few, many); "consumer-supplier" (many, few); and "worker-colleague" (many, many).



1. Director vs Manager;
2. Manager vs Worker;
3. Worker vs Colleague;
4. Consumer vs Supplier;

Figure 4.11: Conceptual Role Pairs

The invocation times indirectly reflect a particular module's involvement in the contribution to system functionality. We use fuzzy concepts "few" and "many" to symbolize the invocation frequency. It reveals how much computation this module has

carried out to undertake the whole work of the specific system functionality. "Few" says that this module generally dispatches the work load to its partner modules, it is in charge of management work; "Many" reveals that this module has taken a considerable work load portion of the system functionality, it is in charge of the implementation part of system functionality.

We have designed four role-relation pairs and eight roles to reveal the relationships among code modules, they are: (i) Director vs Manager (Few-Few); (ii) Manager vs Worker (Few-Many); (iii) Retailer vs Wholesaler (Many-Few); (iv) Worker vs Colleague (Many-Many).

Collaboration Investigation. This component will generate the query results for related collaboration patterns. We can produce the collaboration pattern list which contains a set of selected routines, and discover their features by exploring the collaboration pattern components. The selection of a sender module or receiver module in order to find out the other related parts has similar effects. We can use any combination of these four elements (pattern, sender module, receiver module, and routine) to generate the list of the remaining elements.

Role Investigation. This component explores the role space of each module in a selected collaboration pattern. If we assign to the sender module a certain conceptual role, we will be able to retrieve the modules that have the correspondent role in a given collaboration pattern. We are able to compare roles for a single module in different patterns; produce the dominant role for a particular module based on the multiple role information that it has carried out in different collaboration patterns.

With the help of visual expression provided by *Dyanmic-Anaylzer*, and the analysis components provided by *Collaboration-Investigator*, we are now able to generate comprehensive system functionality construction structures to reveal partial system architecture.

Figure 4.12 illustrates the system module constructional structure that implements one sub-system's functionality in a sample case study. It exhibits the inter-relationships

among all the modules that contribute to the implementation of one specific system functionality. The integrated diagram can be further decomposed into a set of visual representations of collaboration patterns, and the assignments of the major role for each module.

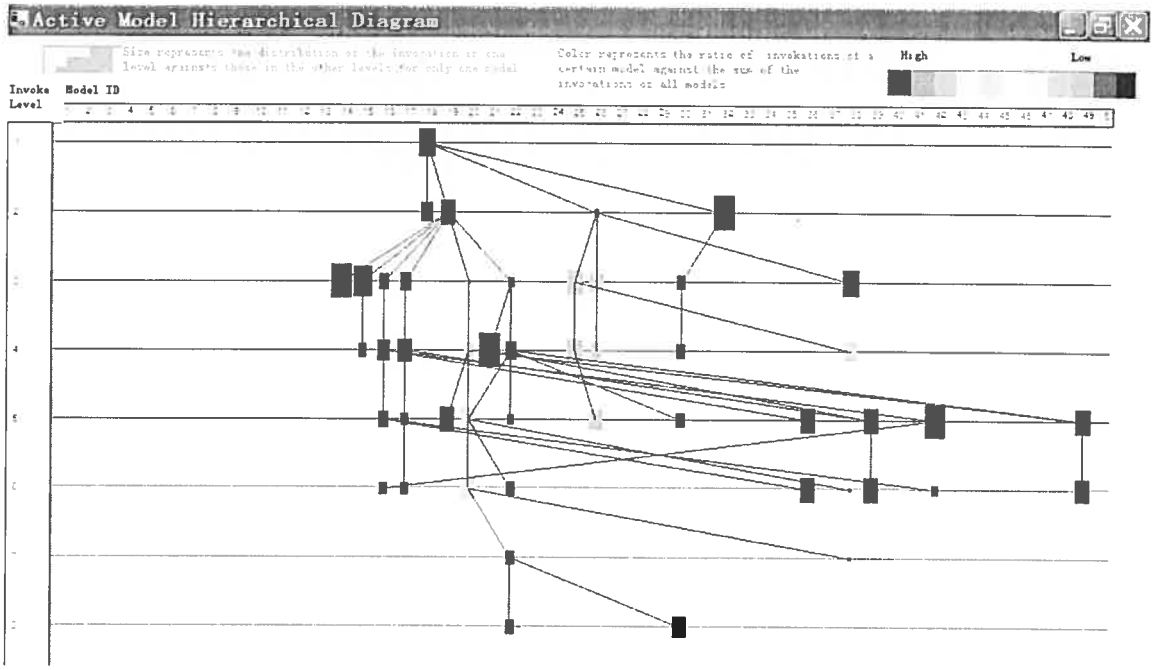


Figure 4.12: Partial System Constructional Structure Visualization

4.1.7 Discussion

In this section, we have presented an approach to recover collaboration patterns and roles from legacy systems for the purpose of legacy software understanding and system decomposition. It consists of two major parts, both of which are supported by our reverse engineering tools. The first part focuses on the dynamic analysis of target legacy systems, and the automatic discovery of collaboration patterns. The second part concentrates on the recovery and analysis of collaboration patterns and roles with human intervention.

Each recovered collaboration pattern represents a concrete implementation block of the observed system functionality. By characterizing such kind of program construction,

we gain the insight of how legacy system behavior is carried out through the collaborations of its source code. It is also useful to apply the discovered collaboration patterns to a further decomposition of the whole system into a role-based hierarchical representation. The maintainer can use this information to study each module within various collaboration patterns, thus to regain more detailed source code modularization information. Cohesive measurement is also used to perform legacy reconstruction [Van00][Kui00]. Within a collaboration pattern, its composition modules intensively cooperate together to perform a concrete task inside of the system functionality. Therefore, collaboration pattern recovery can be further used in the re-modularization of legacy system.

The recovery of collaborations provides us with a decomposition view of legacy software. Most work on understanding interactions has focused on visualization techniques, where the challenge is to develop efficient ways to visualize the large amount of dynamic information [Wal98][Sys01]. The work of DePauw et al. [Pau98], now integrated with Jinsight, allows engineer to visually recognize patterns in the interactions of classes and objects. ISVis displays interaction diagrams using a mural [Jer97] technique. Our work in the visualization part is similar to these two approaches. Instead of the mere focus on visualization, our approach emphasizes more on the recovery of collaboration and the understanding of roles. Tamar et al. also propose an approach to analysis of roles within collaborations [Ric02], but they purely use the invocation methods as representative of roles. This is not sufficient in our research to analyze the general function of a module inside of recovered collaboration pattern. We use predefined conceptual role stereotypes for the recovery of roles based on the invocation relations with other modules.

4.2 Legacy System Decomposition

The system decomposition is important to limit the complexity and risk associated with the re-engineering activities of a large legacy system. It divides the system into a collection of meaningful modular parts with low coupling, high cohesion, and minimization of interface, thus to facilitate the application of our incremental approach to

implement the progressive migration process. We have designed three major techniques to decompose legacy systems. We present them in detail in this section.

4.2.1 Visualization and Dynamic Analysis of System Functionality

System usability is embodied in detailed atomic system functionalities, which represents the concrete utility of the system. We design this visualization and dynamic analysis technique to build up the linkage between legacy source code construction and system functionality. It is designed to further facilitate the legacy system decomposition based on system dynamic execution information. Our approach is demonstrated as following. A modified legacy source version is generated by injecting probe code into original source code. During the execution of a legacy system with detection code, we are able to retrieve the dynamic information of specific system functionality by performing a range of test cases. Meanwhile, we use visual effects to link system behavior with legacy source code, and further generate the visual diagram to reflect the structure of the observed system functionality.

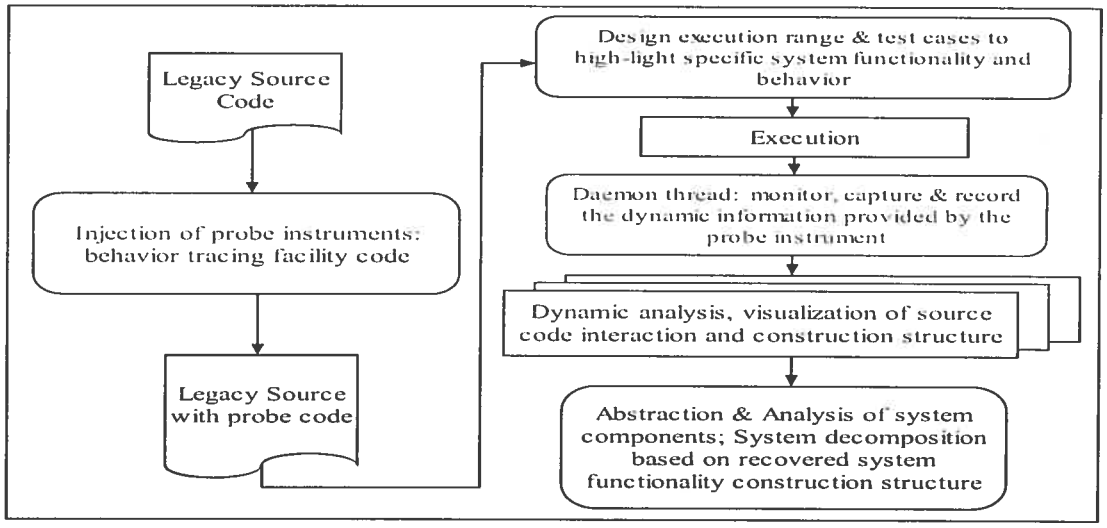


Figure 4.13: Visualization and Dynamic Analysis of System Functionality

The system decomposition task will be performed based on the recovered system functionality source code construction structure. The system functionality-based decomposition approach is illustrated in Figure 4.13. Visualization is implemented at different levels based on different granularity of abstraction.

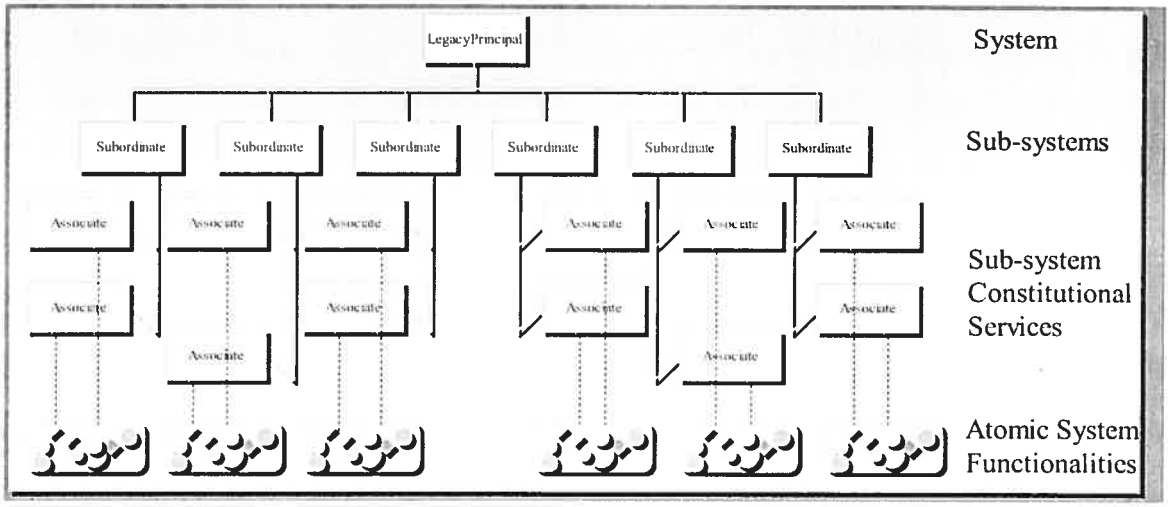


Figure 4.14: Hierarchical System Functional Abstraction View

As we have discussed in the beginning of this section, system utility is performed by those detailed atomic system functionalities, which embody the service of the system. The executable system can be viewed in a hierarchical manner as: whole system, subsystems, subsystem constitutional services, detailed atomic system functionalities for each service, etc. (see Figure 4.14).

The physical program source code can also be viewed in a hierarchical way as program module layer (source code files/modules), and source code entity layer which includes data structure components, database components, functional component (variables, functions, procedures, routines, data, etc.,) (see Figure 4.15). We try to answer the following question: "Which program artifacts realize the observed atomic system functionality that is provided by the legacy system? And how?"

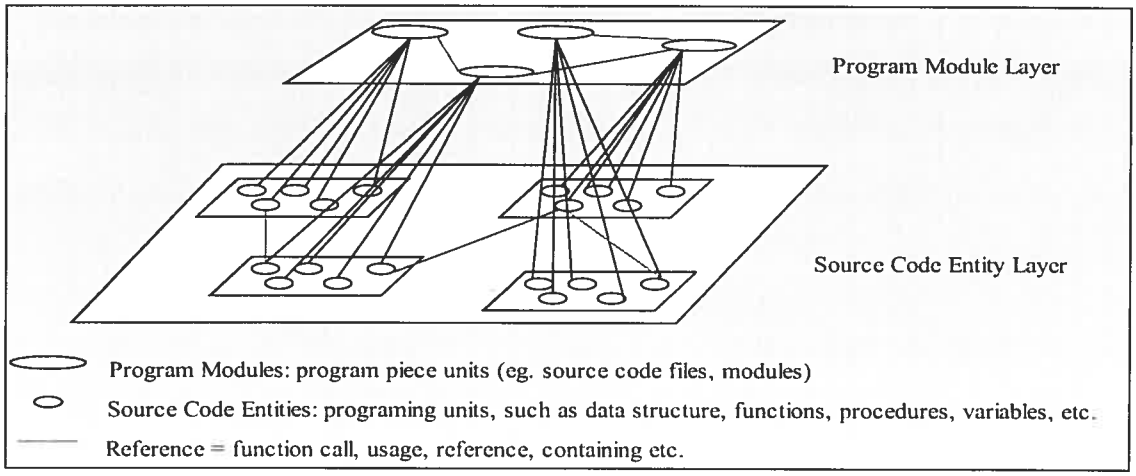


Figure 4.15: Source Code Hierarchical Abstraction View

Mapping atomic system functionalities to the realizing code fragments, and recovering the interaction relationships among source code artifacts will divide the whole source code into constructional parts, thus reveal the structure of legacy software. Based on this approach, we have developed our dynamic and visualization analysis technique to analyze active system functionality behavior, and build up the linkage between the corresponding code artifacts, their interaction structures, and observed system functionalities. One of the most important features of this technique is the mapping ability for different abstraction layers. The results could be seen with the following views:

- **Routine Interaction View:** it shows the inside of source code module, and illustrates which routines (program functions or procedures) interact with each other to contribute to the performance of a certain kind of atomic system functionality.

- **Module Interaction View:** it demonstrates which source code modules work together to carry out a certain kind of system functionality (see Figure 4.16). This information will be used to facilitate the system decomposition process.

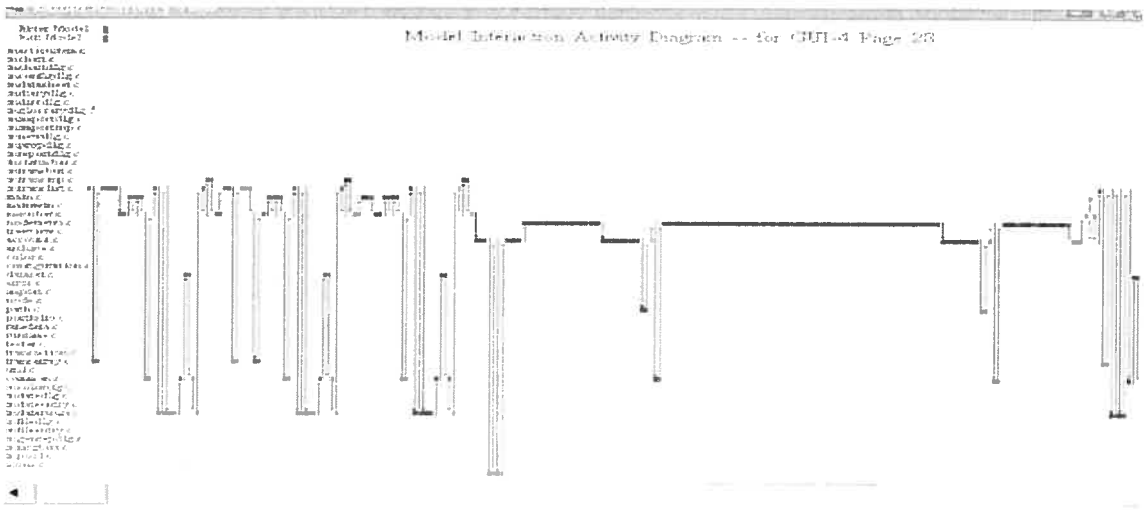


Figure 4.16: Module Interaction View

- **Module Contribution Comparison View:** The comparison view of two modules provides the visual comparison of any two source code modules (see Figure 4.17). It is used to analyze the detailed efforts of modules at each invocation level. The vertical coordinate represents the number of invocation times; the horizontal direction shows the invocation depths. We use cardinal splines to diminish the sharp angles of the curve. This method creates some drawing under the horizontal coordinate to reduce the radical changes from high value to zero.

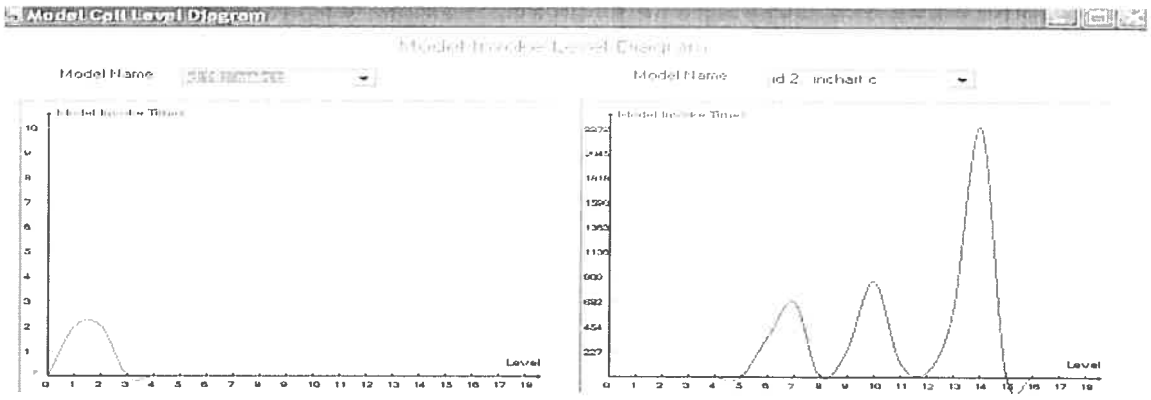


Figure 4.17: Module Contribution Comparison View

- **Module Participation View:** It is provided to analyze the overall participation percentage of each source code module for the observed system functionality (see Figure 4.18).

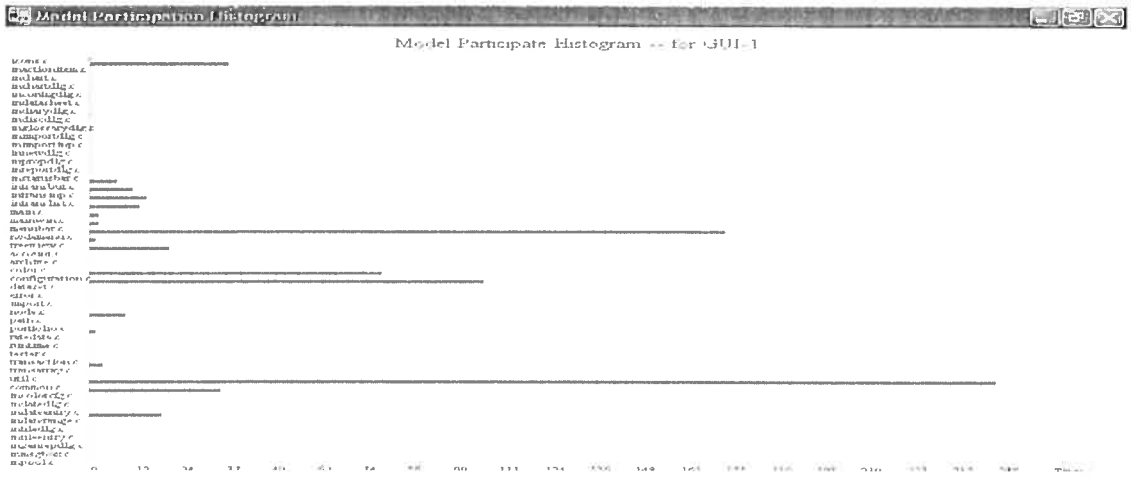


Figure 4.18: Module Participation View

The vertical coordinate indicates the name of each source code module; the horizontal axis shows the scale of invocations. The histogram demonstrates the overall efforts of each module that contributes to the implementation of the observed system functionality. The view shows the difference of contribution for each source module during the execution of specific system functionality.

- **System Functionality Construction View:** Normally, a mapping between fine-grained code artifacts and system behavior will make the maintainer lost inside of the huge size of code interaction space, and limit the usability of our technique. By mapping a particular system functionality with different abstract layer of program artifacts, a more efficient high level view of system structure will be revealed. Consequently, we are able to discover the relationships among different system functionalities and corresponding program artifacts at various abstraction levels. The result is later used to facilitate the legacy system decomposition task.

- **Statistic Information Analysis View:** For a certain kind of system atomic functionality, by executing different test cases, we are able to get different sets of code fragments at different abstract layers. Our dynamic analysis tools are used to retrieve the following statistic information: which parts of each different abstract layer has persistently participated; which parts are conditionally involved; which parts undertake the heaviest computing part, and which parts mostly contribute to the task dispatching and management job, etc.

- **Fine-grain Detail Code Entity Exploring:** By using traditional reverse engineering tools, we can obtain useful information deep inside of legacy source code, thus to ease the system decomposition task. The source code information includes the parsing result of AST (abstract syntax tree), the data-flow diagram, the routine, variable and data structure reference graphs, etc. In our research, we use reverse engineering tools "Source-Navigator" [Gnu00] (see Figure 4.19), to parse legacy source code and generate various intermediate result information.

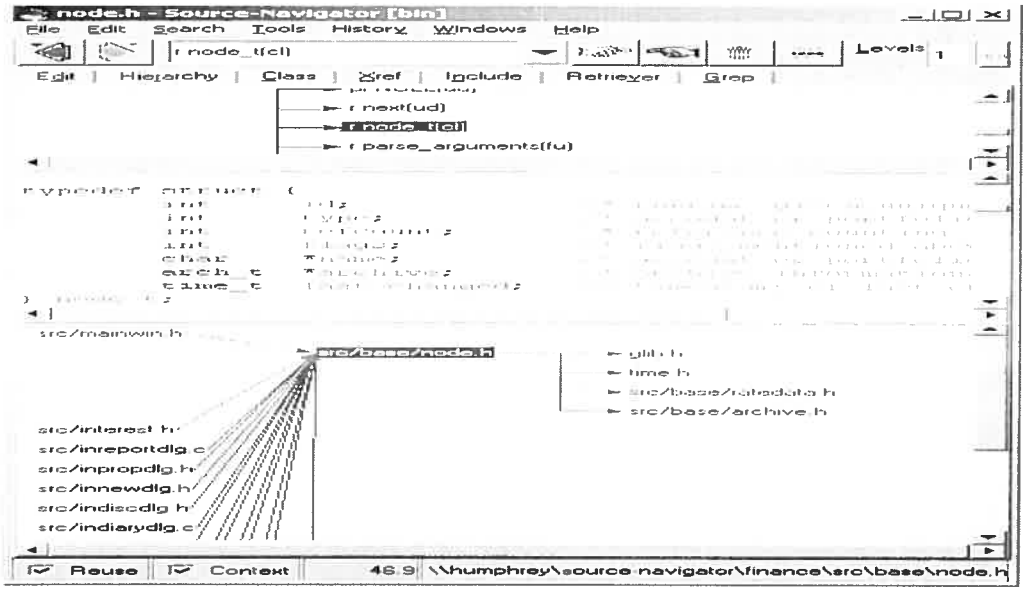


Figure 4.19: Variable, Routine, and Macro Reference Graph by Source-Navigator

Decomposition Strategy with Program Visualization and Dynamic Analysis

Recovering the architecture of legacy systems requires more than just reverse engineering tools to generate some descriptive diagrams of systems. Software architecture is commonly defined in terms of components and connectors. While based on procedural methodology, there is no concrete concept of components and connectors in legacy systems. We have to abstract the high-level architecture based on its component-like parts. Therefore, by dynamically analyzing atomic system functionalities with software visualization, our decomposition strategy is to separate the source code artifacts based on their involvement of a certain type of system functionalities. For those code modules that have participated in more than one system functionality, they will be separated to form a new partition part which serves other functionalities. With the help of our visual dynamic analysis technique, we will later be able to decompose the whole program artifacts into cohesive parts which will reveal the system constructional organization.

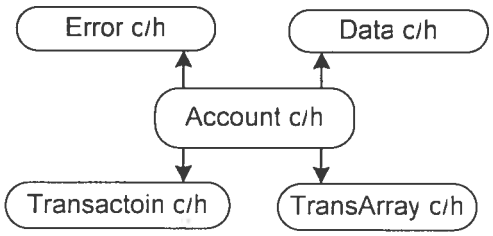
4.2.2 Decomposition with Module Dependency Analysis

For most legacy procedural languages, the whole system can be divided into program pieces (such as source code files) [Bal01]. We look at such individual program unit as a single source code module. Normally, original developers had a certain kind of principle to organize their program artifacts. The dependency of user-defined data types (UDT) reflects the module relations between each other [Van00]. It can be further viewed as an indicator of the associations between the host modules and the rest of the system. By analyzing the dependency of user defined data type, we can elicit useful design information of legacy system architecture construction. For example, in a source code module, called Account.c/h, there's a User Defined Data Type:

```
typedef struct _ac_t {
        node_t  node;           /* has to be the first member (inherit)      */ ---Error.c/h
        char    *symbol;        /* symbol or id                              */
        char    *altsymbol;     /* alternative symbol (for import)           */
```

```
        prec_t    precision;        /* precisions for in/output                  */ ---Data.c/h
        double    fraction; /* fraction of an account (factor)                   */
        GPtrArray *transarr;        /* GArray<trans_t> transaction list          */ ---TransArray.c/h
        int       flags;            /* frozen == readonly                        */
        period_t lifetime; /* time period: 1st to last transaction               */ ---Transaction.c/h
        GArray *rtarr;              /* GArray<rt_t>: "prepared" trans data       */ ---TransArray.c/h
} ac_t;
```

(i)



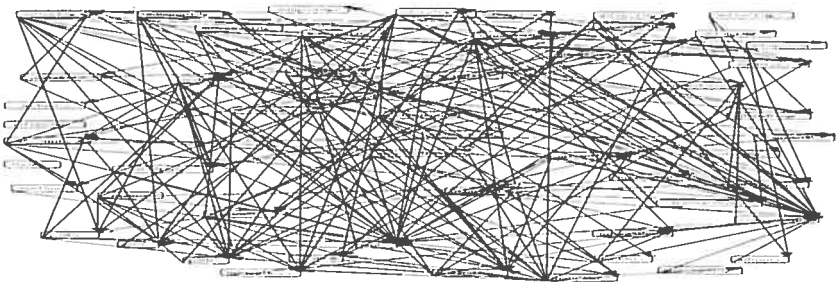
(ii)

Figure 4.20: Source Code Module Dependency Analysis

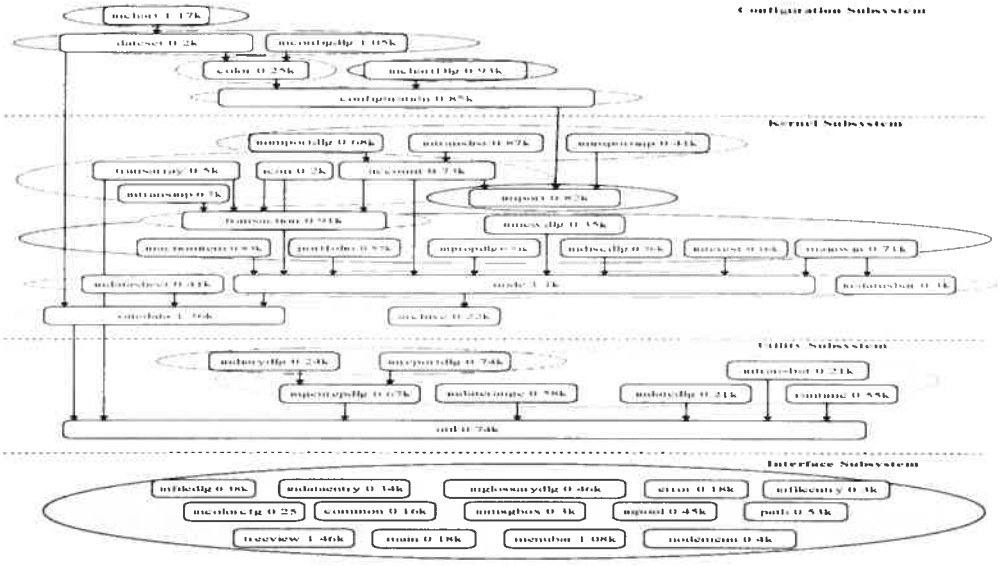
The user data type dependency (one UDT uses other UDTs) can reflect the reliance relationships between its host module and its dependent modules. To explain this, we use the example illustrated in Figure 4.20. The model dependency diagram (Figure 4.20 (ii)) is generated from the source code of a user defined data type (Figure 4.20 (i)). Source code module Account defines one UDT called _ac_t, which uses four other UDTs defined in other four modules, Error, Data, Transaction, and TransArray. The relationships reflect the dependency between Account module and the other four modules. Those modules involved in this relation will be granted higher coupling value than others which do not have such relation. We can iterate this process to assign coupling degrees among each module pair. Therefore, the module dependency relationship can further be viewed as an indicator for dividing the whole system source code into organically integrated parts.

The borders of construction parts are the borders of the module dependency relationships. Consequently, according to the module dependency relations, we can decompose the whole system into parts. We have devised our second system decomposition approach based on source code module dependency analysis. Figure 4.21

illustrates an example of decomposing a legacy system based on this approach. The first diagram of a call graph (Figure 4.21 (i)) demonstrates the interaction relationships among all the source modules. Eventually, this type of information becomes too overwhelming to comprehend: it is hard to distinguish which modules have higher coupling features, which do not, thus it makes it difficult to decompose the whole system based on coupling and cohesion analysis. The second diagram (Figure 4.21 (ii)) is constructed based on module dependency analysis. It further divides the whole system according to module dependency relationships. The lines represent the dependency relation.



(i) Legacy System Module Call Graph



(ii) Legacy System Decomposition: Module Dependency Diagram

Figure 4.21: Source Code Decomposition with Module Dependency Analysis

4.2.3 Migration Unit Construction and Decomposition Algorithm

Modules will be clustered into groups that can be migrated at one step of the process, i.e., processed by a single team within reasonable time limits, called *Migration Units (MU)*. To fix *MU* dependencies between modules, e.g., service delivery, a *dependency graph* is constituted where vertices are modules, and the service requests (invocations) are directed edges. There are two types of module dependency relationships. One is the function call dependency relations between host module and other dependent modules; the other one is the module dependency relation defined by user defined data type, which is presented in section 4.2.2. We give higher coupling value to the later one since this type of coupling (dependency) relationship is defined at the design level which reflects the source code static structure. An intermediate step is the factoring of that graph into s*trongly connected components (SCC)* (i.e., maximal groups of modules that depend on each other, either directly or by transitivity), and the construction of the respective *directed acyclic graph (DAG)* of all *SCC*. The *MU* then emerges as subsets of the *levels* in the *DAG*. Here, a level m is defined as the set of graph nodes for which the longest path from (one of) the universal source node(s) (no in-going edges) in the DAG is of length m.

Finding MUs in Legacy Software

A key task is the split of the entire system into tractable parts which are both (i) logically connected and (ii) admit a single-step migration. As a first approximation that fits (i), one may identify MUs with modules. However, most modules are of smaller size so that a module-based decomposition may lead to a large number of individual migration tasks and, thus, to a high synchronization cost. Therefore, it is more convenient to define MUs as sets of modules, whereby a *preferred size* for MUs becomes an important parameter of the decomposition that insures (ii). However, condition (ii) imposes some restrictions on the way modules are grouped into MUs. In fact, whenever two modules A and B are linked in the graph, say A depends on B, or $A \rightarrow B$, it is preferable to migrate B before migrating A in order to avoid some reworking of the

already migrated code imposed by the decision *A* before *B*. This means that if the duration of migration tasks is a concern, one should consider (ideally) only MUs made up of modules with no dependencies. Such groups may be identified by looking at the module dependency graph, as defined by previous section, where candidate MUs are independent subsets of graph nodes. Unfortunately, in a large number of cases, the information provided by the graph may not be sufficient. Indeed, the strict respect of inter-module dependencies may be hindered by the presence of graph circuits, i.e., configurations in which sets of modules depend on each other, either directly or by transitivity. In this case, neither of the concerned modules may be migrated before the others without some post-reworking. The solution seems to reside in the joint migration of all the modules. However, the resulting task is of a much higher complexity than the migration of an equal number of unrelated modules. Therefore, the number of such tasks should be kept to a strict minimum. To formalize the underlying problem, we apply a graph-theoretical framework based on the concept of *SCC*. The approach is summarized as illustrated in Table 4.2:

```
Build module dependency graph G;
Apply discovery algorithm to detect the set S of all SCC and to organize S in the DAG D;
i = 0;
while S is not empty
    Level[i] = all nodes in D with no in-going edges (no service to other nodes);
    Erase the nodes of Level[i] from S and D;
    i++;
for j from 0 to i-1 do
    if ( the size of each node in Level[i] < threshold size) then
        Group nodes into a minimal set of MUs so that each MU size < the threshold;
    else
        for each node whose size > threshold size do
            Apply slice-merge strategies to split the node into smaller nodes (modules);
            Group the resulting nodes into a minimal set of MUs of size < the threshold;
Return the set of discovered MUs;
```

Table 4.2: MUs Construction: Decomposition Algorithm

The *SCC* of the dependency graph is considered instead of single nodes, i.e., modules, and the respective factor-graph, the *DAG* of the *SCC*[1], is examined for MUs. The sets of independent nodes in the *DAG* are drown from its level-wise split: the highest level, e.g. *0*, is made up of all nodes (if any) that do not provide services to the outer world (universal clients), while the nodes of level *n+1* are those which provide services only to nodes of levels *0* to *n*. With such decomposition, the MUs may be obtained by partitioning each level into subsets of desired global size. It is noteworthy, in the case of a huge-size module or SCC, that a slice-merge strategy is applied to decompose it into MUs. The process is of heuristic nature, where one possibility is to measure the cohesion of the obtained parts and the coupling between parts. The approach is illustrated by the example of Figure 4.22.

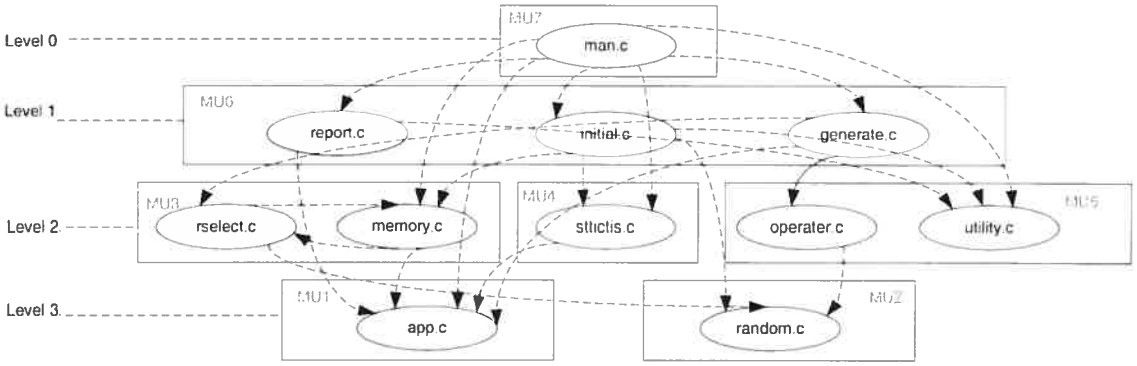


Figure 4.22: The Decomposition of "SGA-C" Legacy System

It shows the dependency graph of the SGA-C system, the Genetic Algorithm C-language port and extension system [Gol89], which is made up of 11 modules. These are first split into 10 SCC, only one of them is non-trivial (made up of rselect.c and memory.c). The SCC is further organized into 4 levels which give rise to 7 MUs (drawn as rectangles around member modules).

4.2.4 Discussion

[1] In the case of no circuits, the DAG is identical to the initial graph.

In this section, we have presented three techniques that we have developed to conduct legacy system decomposition task. They are visualization and dynamic analysis of system functionality, module dependency analysis, and decomposition algorithm, respectively. They are constructed based on different emphasis of system analysis aspects. In the real-world practice, we shall apply all these techniques, and compare, evaluate, and optimize their results to produce the optimum solution. The decomposition result will later be used to facilitate the incremental migration process in the next migration stage.

4.3 Summary

In this chapter, we have presented our legacy system analysis approach in the practical migration model. We focus on two parts, one is the legacy system decomposition, and the other one is legacy source code collaboration pattern/role recovery and analysis. The system decomposition part focuses on how to decompose legacy system into parts, thus facilitates the next stage of applying a divide-and-conquer approach to implement the legacy system Object-Oriented re-architecturing and incremental migration. The decomposition strategies are constructed based on different emphases of system analysis aspects. The code collaboration pattern and role recovery concentrates on the identification of legacy source code features and construction structures. Each recovered collaboration pattern represents a concrete implementation block of the observed system functionality. By characterizing such kind of program construction, we gain the insight of how legacy system behavior is carried out through the collaborations of its source code. The recovery of roles further supports the decomposition of the whole system into a role-based hierarchical representation, thus to provide us with a decomposition view of legacy software.

Chapter V. Object-Oriented Re-Architecturing

The object-oriented re-architechuring of legacy systems generates the object-oriented model of subject system. In an OO system, a program is constructed around data rather than the services of the system, and the supporting structure is a set of classes that are related to each other with different kinds of relationships. In a well-designed OO system, the classes must exhibit high level of cohesion and a low level of coupling. Similarly, in structured programming languages, programs can be seen as sets of data items (variables) and routines that use these variables. Legacy system re-architecturing is therefore achieved by the discovery of object-oriented features in legacy system and the generation of object model for the legacy code. UML is applied to illustrate the modeling result. The key task here is the extraction of classes and their relationships from legacy source code.

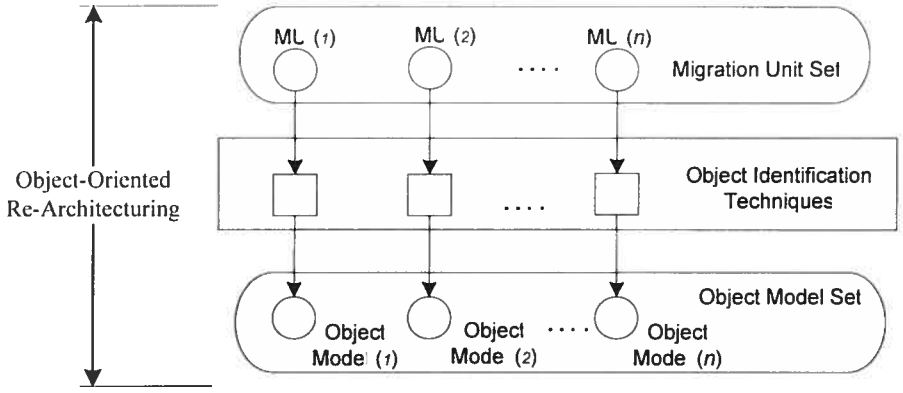


Figure 0.1: Object-Oriented Re-Architecturing

To facilitate this task, we focus on the object-oriented modeling of migration units (decomposed parts), which are generated from the former system decomposition step (see Figure 5.1). We apply several techniques (detailed in the following sections) to extract the object models from legacy systems. The final object-oriented model is hypothesized to have high modularity, high cohesion inside the class, and low coupling between classes.

5.1 Rule-based Class Recovery

Object model discovery can be conducted by using a number of different software analysis techniques. However, no matter how sophisticated those analysis techniques are, users' assistance and guidance are crucial in obtaining a viable and efficient object model. In order to guide the discovery process and to obtain a better object model, as a guideline, we construct a general set of candidate class recovery rules to facilitate the class construction process. Based on the experiments we have conducted, we try to elicit most of these class identification principles, and cast them into an automatic manner in identifying candidate classes. The rules define the criteria of the production of object model from legacy source code, and the goal is to achieve high cohesion within a class and low coupling between classes.

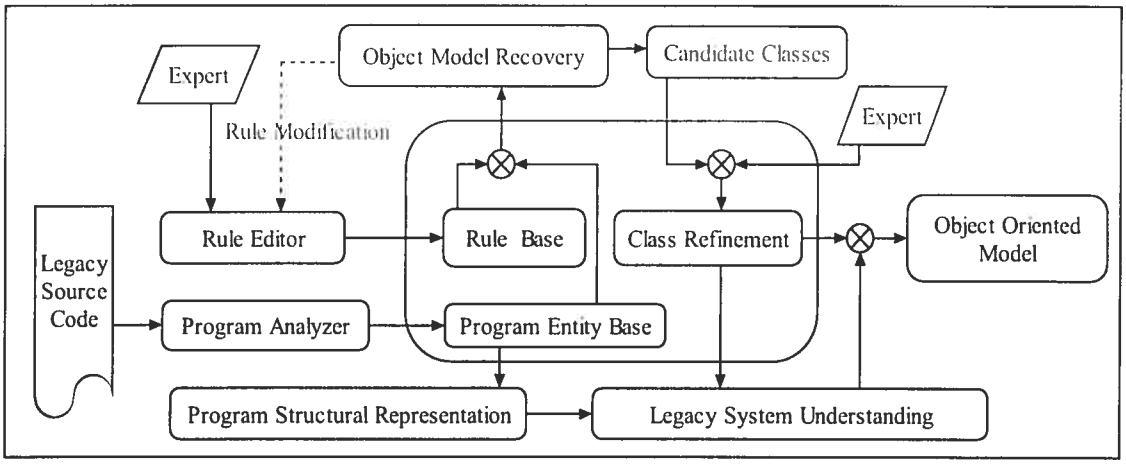


Figure 5.2: Rule-based Class Recovery Process

We document these class recovery rules that human experts applied in identifying classes from procedural programs, and put the class recovery rules into rule-based repository. During the process of class recovery, these recovery rules will be iteratively refined by the feedback of the results. The approach is illustrated in Figure 5.2. More importantly, several rules can be used together to discover potential classes in a semi-automatic way. The correctness of a certain rule and its applicable scope will be calculated by experimental analysis. Therefore we should be able to evaluate the performance of our class recovery rules, and assess the accuracy and the applicable scope. Consequently, object identification tools may reuse these class recovery rules to discover classes in a more automatic manner.

5.1.1 Terminology

To better analyze the legacy system, we have the following definitions:

- *Routine:* is either a function or a procedure in a legacy system. They are the atomic behavior units which interact with each other to perform a certain kind of legacy system functionality.
- *Variable:* is the data item carrier in a program. There are two main scopes of variables in a program: local scope and global scope.
- *User defined data type (UDT):* is a data type that is defined by the user. It normally includes a collection of variables with host programming language build-in data types. It performs as an independent entity data type. For example, in procedural language C and Pascal, "struct" and "record" are constructs that can be used to generate user-defined data type structures.
- *Super-type and sub-type:* if data type X is used to define data type Y, then we say that X is a sub-type of Y, and Y is a super-type of X.
- *Global variable:* is a data item carrier that has program range effectiveness. This mechanism allows data entities to be treated throughout the whole program.

5.1.2 Candidate Class Discovery

Rule_1: *Each UDT can be converted into a candidate class.*

User-defined data type (UDT) refers to a collection of variables that are grouped together, and they can be viewed as a unity. This resembles Object-Oriented design where the related data variable items are grouped together in a class. Therefore, we can view each UDT in procedural program as a candidate class. Each variable in the UDT can be converted into the corresponding attribute of the candidate class.

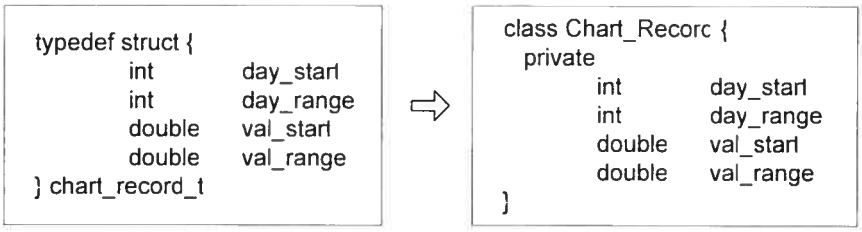


Figure 5.3: Converting UDT into a Candidate Class

Figure 5.3 shows an example of converting a UDT into a candidate class. This UDT defines a new data type called chart_record_t which is used to store the record information of stock analysis chart. It includes four variables: the start day, the range of the days, the start stock value, and the range of the stock values. From OO design point of view, this user-defined data type can be seen as a concrete programming unit: all the stock chart analysis work is conducted based on this data type. Therefore, we can convert this UDT into a candidate class, called class Chart_Record. Moreover, for those functions that mainly deal with this UDT can be further included into this class as members.

Rule_2: *If a UDT (super-type) includes other UDT (sub-type) as its data item, we can assign a composition relationship among the converted candidate classes which are obtained from Rule_1.*

When one UDT contains other UDTs as its variable items, it can be viewed that the host super-type UDT is composite of the variables that are defined by other different sub-type UDTs. When we apply Rule_1, these UDTs are all converted into classes. Then, we

can see that the host class is composed of other classes. Figure 5.4 illustrates this concept with an example. The host super-type UDT _account_t has five variables that are defined by five sub-type UDTs. After we converted all these UDTs into candidate classes, we thus get composition relationships between the host class and the data item classes.

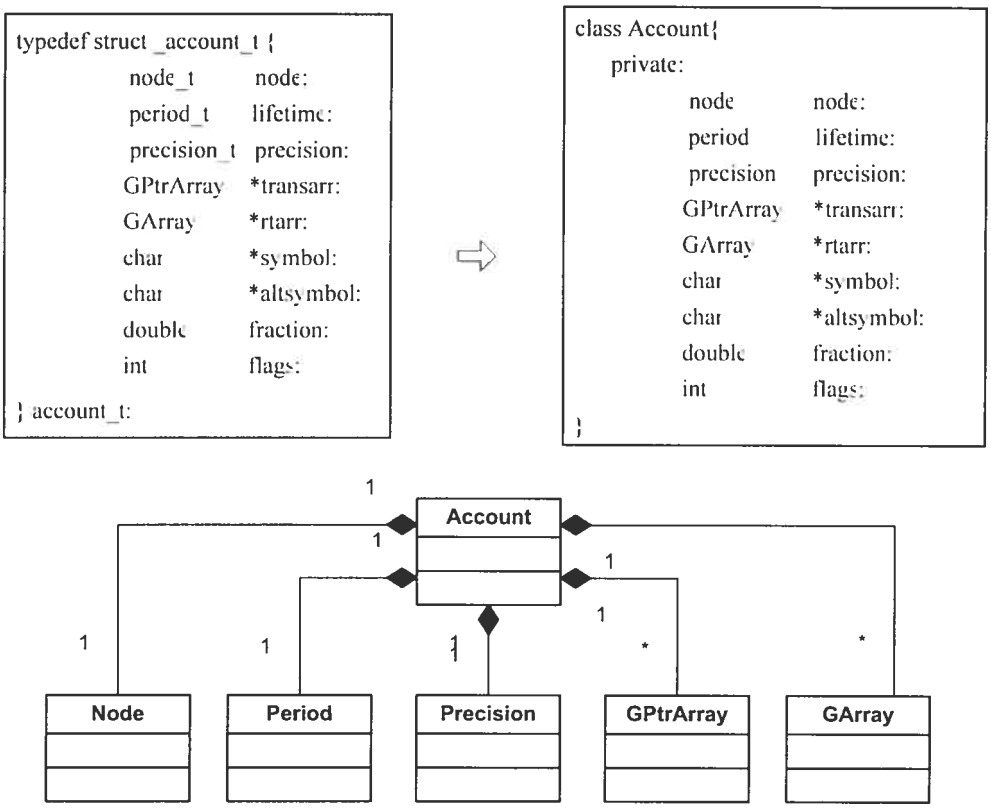


Figure 5.4: Deriving Composition Relationship

Rule_3: *If there is a unique UDT which is a routine's parameter's data type or return value's data type, then this routine can be converted into a member function of the class which is derived from that UDT.*

Parameters and the return value of a routine work as input and output of that routine. It indicates that the routine modifies the data items of the UDT. In the process of object model extraction, we consider routines as method candidates. To maximize the cohesion inside the class and minimize the coupling between classes, the routines with parameter's data type as UDTs are attached to the candidate classes that are generated from these

UDTs. As shown in Figure 5.5, ac_update_runtime, ac_write and ac_show are three routines that use account_t UDT as their paramenter's data type. These three routines access/modify the data fields of the user defined data type of account_t. As a consequence, they are assigned into the candidate class "Account" which is generated from UDT account_t.

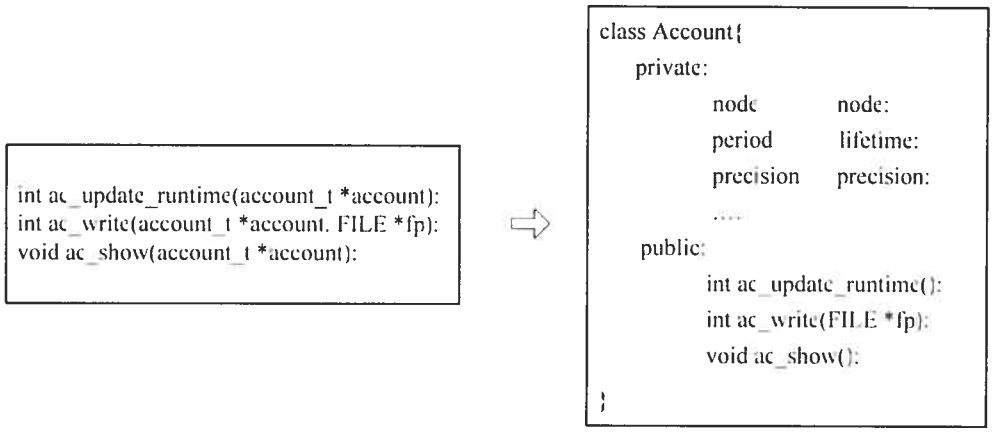


Figure 5.5: Attach Routines into Candidate Class by Parameter Data Type

The return type of a routine indicates that the routine generates or updates the data fields of the UDT of the return value. Since a class should encapsulate methods that update the state of objects that belong to this class, the return type provides strong evidence to include such routine to the candidate class derived from the UDT of the return type. An example is depicted in Figure 5.6.

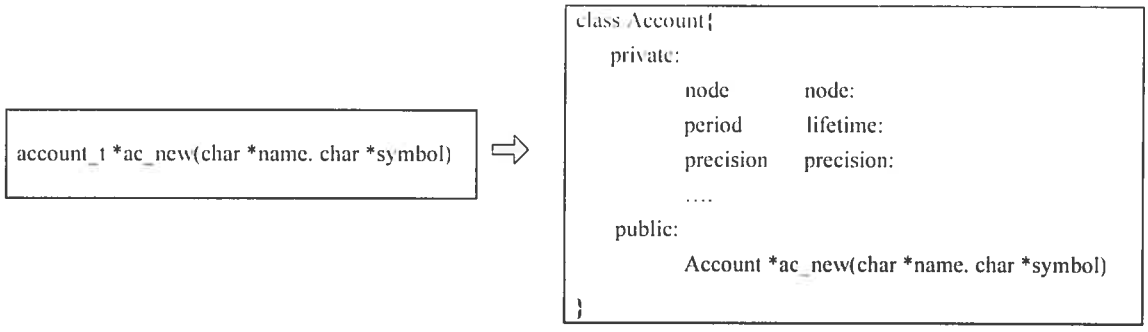


Figure 5.6: Attach Routines into Candidate Class by Return Value Data Type

The routine ac_new() has two parameters: name and symbol. It generates a return value with account_t UDT. These indicate that routine ac_new() has a tight relationship with candidate class Account which is converted from account_t UDT. Routine ac_new() works as if it is part of the candidate class. Therefore, function ac_new can be attached into Account.

When one routine's parameters have more than one UDT as their data types, programmer intervention should be introduced. The routine will be analyzed to see which UDT is the major data type that it is using. The routine will be attached to this main UDT. For example, in Figure 5.7, routine ac_append_trans() has two parameters defined by two UDTs, namely account_t and transaction_t. Inside the routine body, it has invoked two other routines: ac_append and ac_has_changed respectively. Both are affiliated with the UDT of account_t. Therefore, we can decide that account_t is the major UDT that routine ac_append_trans() is associated with.

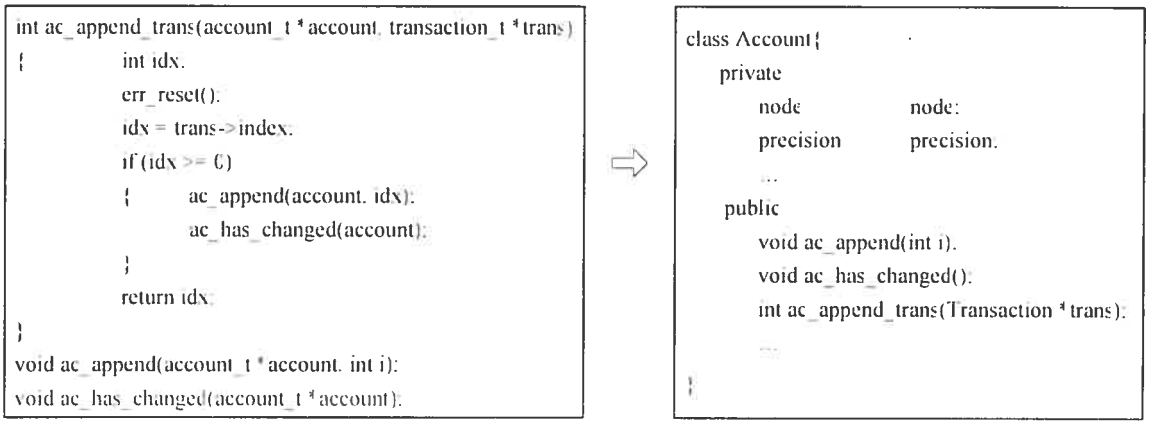


Figure 5.7: Assign Routine to Class When Routine's Parameters Have More than One UDT as Data Types

When a routine does not have UDT as the data type of its parameter or return value, the frequency of the usage of a UDT in the routine's body can be considered as evidence to convert this routine into a method of the candidate class which is generated from that UDT. If there is no involvement of any UDT at all, we can apply the following Rule_4, or static featuring technique which will be detailed in the next section.

Rule_4: *A global variable can be viewed as a candidate class. If a routine uses that global variable, the routine can be transformed into a member function of that candidate class, and the global variable can be converted into an attribute of that class. If a routine uses more than one global variable, those global variables can be attached to one class, and be converted into attributes of that class.*

There are two major scopes to access variables in a program, namely local scope and global scope. For a variable that has local scope, its usability is confined within the routine where this variable is defined. For a variable that has global scope, it can be reached by each routine in the program. Therefore, its status can be modified by all routines. Keeping global scope variables in a new object oriented system would violate the principles of encapsulation and information hiding. A possible solution is that each of these variables can be converted into an attribute of the closely related class, where the class's member functions use/modify these variables. An example is illustrated in Figure 5.8.

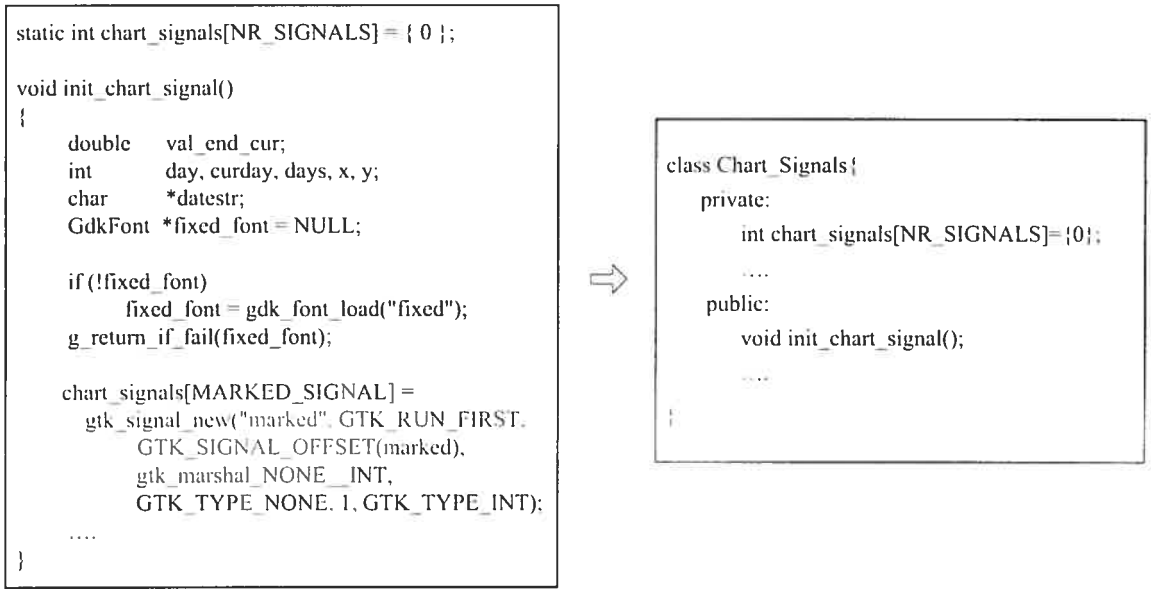


Figure 5.8: Converting Global Variable into Candidate Class

The global variable chart_signals is mainly used by the routines that deal with stock chart analysis. In Figure 5.8, routine init_chart_signal() initializes global variable chart_signals. It assigns values to each of the array item by calling the system function

gtk_signal_new(). According to Rule_4, we can convert the global variable chart_signals into a candidate class, and allocate routine init_chart_signal() as its member function.

Rule_5: *If a UDT includes optional variables inside of the UDT definition body, an inheritance relationship can be derived.*

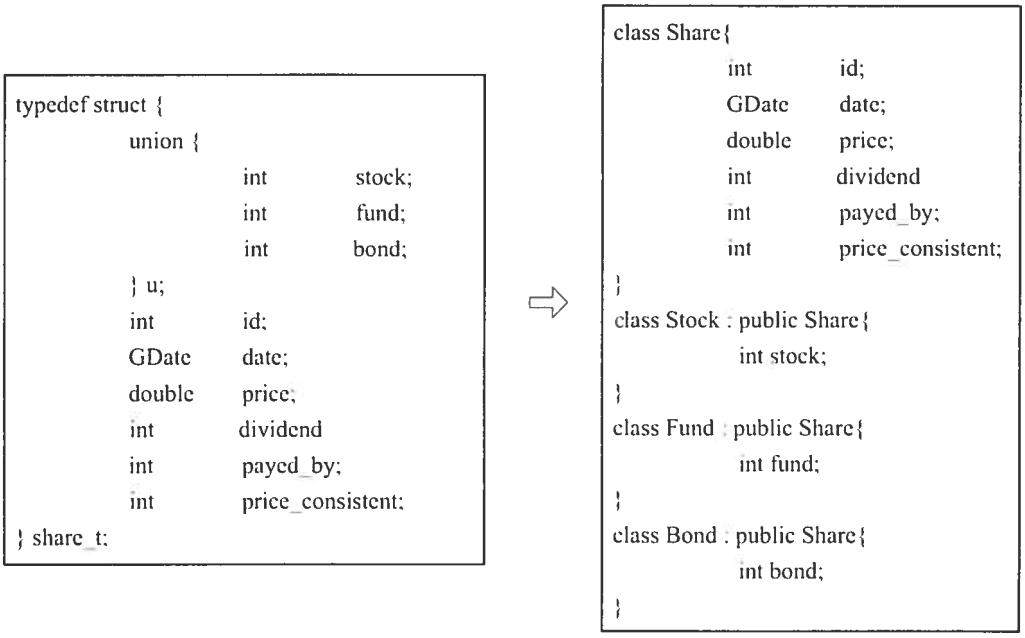


Figure 5.9: Deriving Class Inheritance from Optional Variables in UDT

The optional variables in a UDT can be viewed as variables in the sub-classes which are derived from the original UDT. For example, in C++, the construct "union" defines a user-defined type with optional variables. Only one variable defined in the body can be referenced at a time. In the case that a union type is defined in the body of another UDT, the common structure of the UDT can be extracted as a super-class, while each of the union variables can be assigned into a subclass.

For example, in Figure 5.9, the UDT share_t contains a union type variable. We can derive a super-class called Share from the original UDT which only contains the common variables, such as the id, date, price, etc. Each optional variable in the union definition becomes a subclass, such as Stock, Fund, and Bond.

5.2 Static Featuring Technique

There are mainly two program analysis techniques: static analysis and dynamic analysis. The former one uses source code itself to derive desirable information; whereas the latter one observes the execution of program to gain insights of the subject system. By analyzing program static features, we can derive an object model from legacy source code based on high cohesion inside of candidate classes and low coupling among candidate classes. In collaboration with two members of our legacy migration project team (Idrissa Konkobo and Shiqiang Shen), we have developed two static featuring techniques by using genetic algorithm and conceptual clustering [Sah02][Kon03][She03] to facilitate the automated object identification process. The content of this section is to introduce these two techniques.

The primary working mechanism of these two techniques is based on the discovery of candidate classes as strongly correlated collections of data items through groups of processing code units – the routines. The "correlation" within the tentative objects is evaluated through measurements derived from design quality criteria of coupling and cohesion.

5.2.1 Measuring Class Cohesion and Coupling Metrics

To apply the genetic algorithm and conceptual clustering techniques in object identification task, we have to define the subjective measurement of class cohesion and coupling metrics. To do so, we first define the *completeness* and *consistency* of a strict partition of a given set.

For any set $\boldsymbol{E}$ of entities, we can divide the whole set into a bunch of disjoint sub-sets of $\boldsymbol{E}$. Each element e_i in $\boldsymbol{E}$ can be assigned to a particular sub-set. These sub-sets together form a set of disjoint groups G_i that firmly split the whole set $\boldsymbol{E}$. Formally, we define the *completeness* and *consistency* as illustrated in formula 5.1.

$$\cup G_i = \boldsymbol{E} \qquad (completeness)$$
$$\forall \text{ i and j}, G_i \cap G_j = \varnothing, \ \text{i} \neq \text{j} \qquad (consistency) \qquad (5.1)$$

For the problem of object identification in legacy systems, we primarily have two kinds of source code artifacts. One is the set of data items – the variables; and the other one is the processing code units – the routines. The major goal of object identification is to find the best partition of these two sets to construct an object-oriented model that can yield the desirable high cohesion and low coupling metric values inside and between candidate classes.

Considering a set of n variables $\boldsymbol{V} = \{v_1, v_2, \ldots, v_n\}$, and a set of m routines $\boldsymbol{R} = \{r_1, r_2, \ldots, r_m\}$, there are two types of relations connecting elements in $\boldsymbol{V}$ and $\boldsymbol{R}$. The first relation $VR \subseteq \boldsymbol{V} \times \boldsymbol{R}$, is such that $v_i \ VR \ r_i$ if variable v_i is used by routine r_i. The second one, $RR \subseteq \boldsymbol{R}^2$, is such that $r_i \ RR \ r_j$ if routine r_i calls routine r_j. We further divide $\boldsymbol{V}$ into a strict partition with a set of l groups $\boldsymbol{G} = \{G_1, \ldots, G_i, \ldots, G_j, \ldots, G_l\}$, where each $G_i \subseteq \boldsymbol{V}$. Meanwhile, $\boldsymbol{G}$ should also satisfy the requirements of completeness and consistency defined in formula 5.1. In addition, we consider the set of routines that use the variables in G_i as "the routines of G_i".

Each possible partition of $\boldsymbol{V}$ which is represented by $\boldsymbol{G}$ is viewed as a *solution*. Our goal is to find the best solution which is evaluated by coupling/cohesion metrics to reflect the corresponding OO design quality. Here we give the definition of these two quality metrics.

For any given solution $\boldsymbol{G}$, the *cohesion* metric of a sub-set G_i is defined as:

$$COH(G_i) = \frac{\sum_{k=1}^{n_i} \mu(v_k)}{m_i * n_i} \qquad (5.2)$$

where n_i is the number of variables in G_i, m_i is the number of routines that use the variables of G_i, v_k is a variable in G_i , $\mu(v_k)$ is the number of routines that use the variable v_k. Therefore, the cohesion metric of a specific sub-set of $\boldsymbol{V}$ is basically defined as the number of its routine-variable calls over the number of all possible routine-variable references inside of that sub-set.

The *coupling* metric of a sub-set G_i in solution $\boldsymbol{G}$ is defined as:

$$COUP(G_i) = \sum_j (NRV(G_i, G_j) + NRR(G_i, G_j)) \tag{5.3}$$

where $G_j \in \boldsymbol{G}$, $i \neq j$, $NRV(G_i, G_j)$ is the number of times the routines in G_i use the variables in G_j, and $NRR(G_i, G_j)$ is the number of times the routines in G_i use the routines in G_j plus the routines in G_j use the routines in G_i. For a specific sub-set G_i of a solution $\boldsymbol{G}$, the coupling metric value is therefore measured as the total number of external references from G_i to the other sub-sets of the solution. To reach high quality level of OO design, we are seeking an optimized solution $\boldsymbol{G}$ that each of its sub-set G_i has a high average cohesion value and low average coupling value.

```
#define MAXIM 99
typedef int EL_T;
typedef int BOOL;
EL_T stack_struct[MAXIM];
int stack_point;
EL_T queue_struct[MAXIM];
int queue_head, queue_tail, queue_num_elem;
struct list_struct {EL_T node_content;
        struct list_struct * next_node;
        } list;
main() {/* exploits a stack, a queue, and a list */}

/* functions commented by the referenced variable set */
void stack_push(el) {/* stack_point, stack_struct */  stack_full(), stack_top() }
EL_T stack_pop()    {/* stack_point, stack_struct */  stack_top(), stack_Empty() }
EL_T stack_top()    {/* stack_point, stack_struct */}
BOOL stack_Empty()  {/* stack_point */}
BOOL stack_full()   {/* stack_point */}
Void queue_insert() {/* queue_struct, queue_head, queue_num_elem */ queue_full()}
EL_T queue_extract() {/* queue_struct, queue_tail and queue_num_elem */ queue_Empty()}
BOOL queue_Empty()  {/* queue_num_elem */}
BOOL queue_full()   {/* queue_num_elem */}
Void list_add(el)   {/* list* / list_is_in() }
Void list_elim(el)  {/* list */ list_is_in() }
BOOL list_is_in()   {/* list */}
BOOL list_empty()   {/* list */}
Void global_init()  {/* stack_point, list, queue_head, queue_tail, queue_num_elem */}
void stack_to_list() {/* stack_point, stack_struct, list */ stack_Empty(),list_add()}
void stack_to_queue() {/* stak_point, stack_struct, queue_struct, queue_num_elem,
                queue_head,*/ stack_Empty(), queue_insert(), queue_full()}
void queue_to_stack() {/* queue_tail,queue_num_elem, queue_struct, stack_point,
                stack_struct */ queue_Empty(),stack_push(), stack_full()}
void queue_to_list() {/* queue_struct, queue_tail, queue_num_elem, list */
                queue_Empty(), list_add()}
void list_to_stack() {/* list, stack_point, stack_struct */ list_empty(),
                stack_push(), stack_full()}
void list_to_queue() {/* list, queue_struct, queue_head, queue_num_elem */ list_empty(),
                queue_insert(), queue_full()}
```

Figure 5.10: Procedural code of *Collections* in C

A reference graph derived from legacy code is used to calculate the above two metrics. The graph contains two types of vertices, variables and routines, and two types of edges, VR and RR. If variable vi is used by a routine ri, then we will have an edge linking vi and ri which represents vi VR ri. Similarly, if a routine ri calls a routine rj, we will have ri RR rj, and there will be an edge connecting these two routines.

To better illustrate the application of genetic algorithm and conceptual clustering techniques in object identification task, we apply a well-known example of a C program [Can00] (collections) to demonstrate these two algorithms (see Figure 5.10). The program deals with three types of data structures, namely stack, queue, and list. It first defines the data structures; then it defines the routines of each data structure. For the purpose of better understanding, we mark the global variables used by the routine as comments, and mark the routines that called by this routine in bold fond.

Routine r_i	***VR* of r_i**	***RR* of r_i**
1. stack_push	{a,b}	{3,5}
2. stack_pop	{a,b}	{3,4}
3. stack_top	{a,b}	∅
4. stack_Empty	{b}	∅
5. stack_full	{b}	∅
6. stack_to_queue	{a,b,e,f,g}	{4,16,18}
7. global_init	{b,c,d,e,g}	∅
8. list_is_in	{c}	∅
9. list_empty	{c}	∅
10. stack_to_list	{a,b,c}	{4,12}
11. list_to_stack	{a,b,c}	{1,5,9}
12. list_add	{c}	{8}
13. list_elim	{c}	{8}
14. queue_to_stack	{a,b,d,f,g}	{1,5,17}
15. queue_extract	{d,f,g}	{17}
16. queue_full	{g}	∅
17. queue_empty	{g}	∅
18. queue_insert	{e,f,g}	{16}
19. list_to_queue	{c,e,f,g}	{9,16,18}
20. queue_to_list	{c,d,f,g}	{12,17}

Table 5.1: All VR-relations, and one of RR-relations in *Collections*

Table 5.1 illustrates the VR and RR relations. To be simplified, we number the routines from 1 to 20, and name the variables as letters. Letters a, b, c, d, e, f, g represent variables *stack_struct, stack_point, list, queue_tail, queue_head, queue_struct,* and *queue_num_elem*, respectively.

5.2.2 Genetic Algorithms for Object Identification

Object identification in fact is a problem of searching an optimum solution of the partition and combination of two sets: the variable and routine sets. Different variables and routines are organized to form candidate classes. The quality of the obtained object model is measured by two OO design quality metrics: cohesion and coupling. However, this searching is a difficult task mainly due to the large amount of possible solutions. The cost of exhaustive search is normally too prohibitive to be practical. Genetic algorithms (GA) provide a good alternative to the optimization of grouping problems as stated by Falkenauer [Fal98]. Genetic algorithms use techniques inspired by evolutionary biology, and they are a particular class of evolutionary algorithms that generate new and possibly better solutions based on a set of initial ones. For an optimization problem, genetic algorithms are typically implemented as a simulation in which a set of abstract representations (called *chromosomes*) of candidate solutions (called *individuals*) evolve toward better solutions. The evolution starts from a solution set (called *population)* of initial individuals and happens in generations. The quality of each solution is measured by an objective function (called *fitness function*). In each generation, the fitness of the whole population is evaluated, and pairs of individuals are selected from the current population based on their fitness value. For each pair, two operators, *crossover* and *mutation*, are applied using an associated probability to produce a new pair of chromosomes which form the next generation of population. A selection method is used to prioritize the individuals. The individuals with highest fitness quality values are added to the next generation. The algorithm stops when it reaches a convergence criterion, or a fixed number of generations.

```
i = 0
Create the initial population Π_i
MaxSize = the size of Π_i
Evaluate the population Π_i
BestFit = the Fittest Chromosome in Π_i
TheBestEver = BestFit
WHILE not EndCondition DO
        Create the new population Π_i+1
        Add BestFit to Π_i+1
        WHILE size of Π_i+1 < MaxSize DO
                Select two chromosomes C_1 and C_2
                Cross-over C_1, C_2 with probability p_c to C'_1, C'_2
                Mutate C'_1, C'_2 with probability P_m to C"_1, C"_2
                Add C"_1 and C"_2 to Π_i+1
        Evaluate the new chromosomes in Π_i+1
        BestFit = the Fittest Chromosome in Π_i+1
        IF is_better_than(BestFit, TheBestEver) THEN
                TheBestEver = BestFit;
        i ++
TheBestEver is the final solution
```

Figure 5.11: Genetic Algorithm Generic Pattern

Figure 5.11 describes a generic pattern of a genetic algorithm. Based on the nature of application domain, they only differ in the definition of the chromosomes, the operators, and the fitness functions. In the rest of this section, we present each of these aspects in the GOAL algorithm (Genetic-based Object identification ALgorithm).

The objective of the GOAL algorithm is to discover objects (sub-set of variables) from legacy source code. A chromosome represents a solution that includes a set of objects. Each object is called a *gene*. If we consider the example illustrated in Figure 5.10, a possible solution can be made up of three objects: O_1= {a, b}, O_2 = {g, f}, and O_3 = {d, c, e}. This solution can be symbolized by the chromosome:

a, b	g, f	d, e, c

Crossover Operator

The traditional crossover operator is defined by dividing each of the two initial chromosomes into two parts, and interchanging the pairs of parts from different

chromosomes to form two new chromosomes. However, such classical crossover operation does not fit with our object identification requirement since it may not satisfy the completeness and consistency constraint of the newly generated chromosomes. For example, assume we perform the crossover operation on the following chromosomes:

P_1=({a, b}, {g, f}, {d, e, c})

P_2=({a, b, g}, {f, d}, {e}, {c})

If the cutting point is set to 2, the new chromosomes will be C_1=({a, b}, {g, f}, {e}, {c}) and C_2=({a, b, g}, {f, d}, {d, e, c}). Neither of them is a valid solution: C_1 is incomplete (missing variable "d"), and C_2 is inconsistent ("d" belongs to two genes).

To fit with the application domain of object identification, we have defined a more suitable crossover operator which will not cause the violation of completeness and consistency. The operation is defined as following:

1. Copy the right hand side part of P_1.
2. Insert this piece between the two pieces of P_2.
3. Copy the right hand side piece of P_2.
4. Insert this piece between the two pieces of P_1.
5. Remove from the original pieces all the variables that are in the inserted piece.

The result of the crossover of P_1 and P_2 from the above example is then C_1 =({a, b}, {g, f}, {e}, {c}, {d, e, e}), and C_2=({a, b, g}, {f, d}, {g, f}, {d, e, c}, {e}, {e}).

Mutation Operator

The mutation of a given chromosome consists of a random selection between two operations: *merge* and *split*. The *merge* operation combines two genes into a single one. On the contrary, the *split* operation divides one gene into two new genes. For example, the chromosome P=({a, b}, {g, f}, {d, e, c}) can mutate to P_{m1}=({a, b, g, f}, {d, e, c}) or P_{m2} =({a, b}, {g}, {f}, {d, e, c}).

Fitness Function

As we have defined before, a chromosome is a solution and each gene inside of the chromosome symbolizes an object (group of variables). We use the average fitness of the objects (genes) to define the fitness of a solution (chromosome). The fitness of a solution *S* is defined as:

$$F(S) = \frac{\sum_{i=1}^{N} f(G_i)}{N}, N > 0 \quad (5.4)$$

N is the number of identified objects for the solution S, G_i represents an object in S, and $f(G_i)$ is the fitness of G_i. As we have indicated before, the quality (fitness) of an identified object is measured by the cohesion and coupling metric values of that object. Therefore, to achieve high quality (fitness value), we have to obtain a combined optimization of both cohesion and coupling metrics. We need to transform this two-criterion problem into a single-criterion problem. We apply a classical transformation by defining a threshold-based function that separates the solutions according to one criterion value. It divides the solutions into two sets: acceptable and unacceptable groups. The other criterion is used to compare pairs of solutions within each group. The cohesion metric is normalized between value 0 and 1. A minimally-acceptable cohesion threshold is also defined, which is marked as MIN_COH. Let G_1 and G_2 be two objects (genes) of one solution (chromosome), a suitable function f of an object should satisfy the following requirements:

- If ($COH(G_1)$ and $COH(G_2)$ < MIN_COH) and $COUP(G_1) > COUP(G_2)$ then $f(G_1) < f(G_2)$
- If $COH(G_1) \geq$ MIN_COH and $COH(G_2)$ < MIN_COH then $f(G_1) > f(G_2)$
- If ($COH(G_1)$ and $COH(G_2) \geq$ MIN_COH) and $COUP(G_1) > COUP(G_2)$ then $f(G_1) < f(G_2)$

We have defined the fitness function of any object G_i in a solution as following:

$$f(G_i) = \begin{cases} f_1(G_i) = \dfrac{1}{COUP(G_i)+2} + 1/2 & if\ COH(G_i) \geq \min\_COH \\ f_2(G_i) = \dfrac{1}{4(COUP(G_i)+1)} & if\ COH(G_i) < \min\_COH \end{cases} \quad (5.5)$$

The function f_1 and f_2 has been designed to systematically favor those solutions with higher cohesion metric values. The result of each function ranges from: $f_1 \in [0.5, 1]$ and $f_2 \in [0, 0.25]$, respectively.

We now use our GOAL to carry out the object identification task on the example introduced in Figure 5.10. We use the following parameters: MIN_COH = 65%, 75%,

and 5% for the probabilities of crossover and mutation, and 100 for the number of generations. Figure 5.12 shows some of the scoring chromosomes of the initial population. It has 14 chromosomes that are generated randomly.

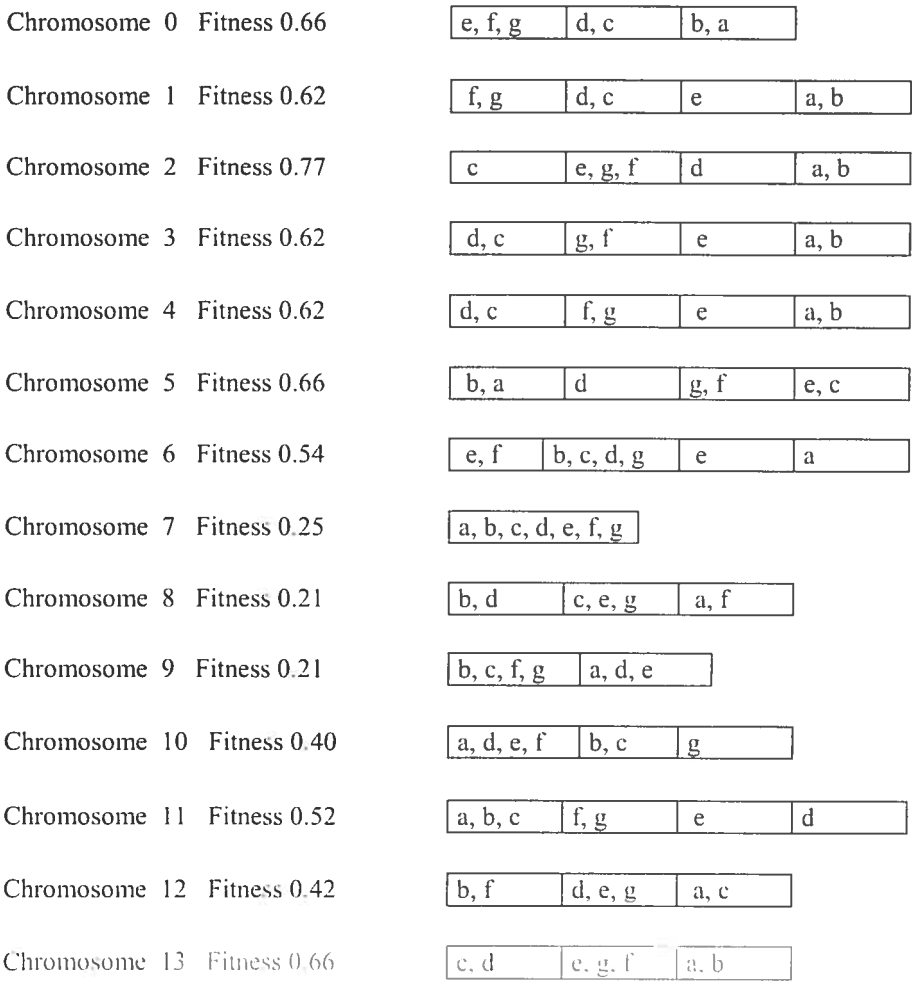


Figure 5.12: The Initial Population

At the beginning, the best solution (chromosome 2) has a fitness value of 0.78. After a few generations, some interesting solutions begin to appear. For example, a chromosome of fitness 0.82 is ({a, b}, {c}, {d, e, f}, {g}). At the end, the best chromosome (fitness = 0.88) was ({a, b}, {c}, {d, e, f, g}). This chromosome contains three genes (objects):

O1 = {c} = {list}

O2 = {e, f, g, d} = {queue_head, queue_struct, queue_num_elem, queue_tail}

O3 = {a, b} = {stack_struct, stack_point}

In fact, we can see that these three objects correspond to the three candidate classes: stack, queue, and list.

5.2.3 Conceptual Clustering for Object Identification

Conceptual clustering [Mic80] [Mic83] is a technique for grouping related items into collections (called *clusters*) in a data set. To solve the grouping problem in the data analysis field, automatic clustering is applied to construct the classification of data items. The clusters contain highly similar items in which inter-group similarities are kept as low as possible. Intuitively, the inner-cluster similarity is interpreted as a reflection of cohesion metric, and inter-cluster similarity is used to represent coupling metric. Based on the measurements of these two metrics, we therefore can apply a conceptual clustering technique to solve the object identification problem: discovering the clusters of global variables as candidate classes in a legacy system.

Conceptual clustering focuses on the direct evaluation of global quality criteria whereas conventional clustering algorithms generally rely on pair wise similarity comparisons. To efficiently conduct object identification tasks, the discovery of conceptual clusters thus can be viewed as following:

Given:

- A set of instances;
- A set of valued attributes to be used to characterize the instances;
- A body of background knowledge, which includes the problem constraints, the properties of attributes, and criteria for evaluating the quality of the constructed classifications;

Find:

- A hierarchy of object categories in which each category is described by a single conjunctive concept. Subcategories that are descendant of any parent category should have logically-disjoint descriptions and should optimize an assumed criterion (a clustering quality criterion).

We have tailored the Cobweb [Fis87], a well-known conceptual clustering method, to build our object identification algorithm. Cobweb uses an incremental procedure to construct a hierarchy of concepts. It applies a hill-climbing search strategy to traverse the space of all possible hierarchies.

Our algorithm C-COI (Conceptual Clustering for Object Identification) is an adaptation of the Cobweb algorithm with a process of iterative optimization to seek a higher quality of the identified objects. Its clustering utility criterion is defined as a compound criterion based on the two quality metrics: cohesion and coupling, which are defined by functions COH (formula 5.2) and *COUP* (formula 5.3) in section 5.2.1. In addition, a set of modified operators are also used to restructure the grouping.

Object Identification Algorithm

C-COI employs an incremental function, OBC (Order-Based Clustering), to discover the hierarchy of variable clusters. It starts from the first variable to form the first single-node hierarchy, and uses a single-variable clustering procedure to carry out the gradual descent of variables through the current hierarchy. It performs a hill-climbing search through the space of all possible solutions, and uses an objective function to evaluate and select solutions. At each concept node, it uses the objective function to evaluate the clustering result from:

- Placing the variable into an existing sub-concept;
- Adding a new concept and placing the variable into it.

In the algorithm, we have set up a limit on the depth of the tree (parameter TREE_HEIGHT), because the obtained tree could be very deep. It is also a proper measure since the node levels indicate the object aggregation levels, which can be logically limited. Moreover, bounding the tree height reduces the cost of the algorithm.

To implement the objective function, we apply a multi-criteria decision-making technique. It consists of a minimal cohesion threshold, MIN_COH, and a subsequent evaluation of the coupling metric. Figure 5.13 gives the definition of the algorithm of OBC (Order-Based Clustering):

```
Given
Type Node = < G: variables; succ: set of Nodes>
N a Node
V a Variable
VS : set of variables

OBC-M(VS) {
  V1 = first variable in VS
  N = create_node({V1})
  FOREACH Vi IN VS-V1 DO
    OBC(N,Vi)
}

OBC(N, V) {
  IF leaf(N) AND level(N) <= TREE-HEIGHT-1 THEN
    N1 = create_node({V})
    N2 = create_node(N.G)
    Add_succ(N,{N1, N2})
  ELSE
    IF level(N) = TREE-HEIGHT-1 THEN
      N1 = create_node({V});   Add_succ(N,{N1})
    ELSE
      FOREACH Ni in N.succ compute cohesion of Gi* obtained by adding V to Ni.G
      Cand = all nodes Ni witht COH(Gi*) >= MIN_COH
      IF Cand = Ø THEN
        N1 = create_node({V});  Add_succ(N,{N1})
      ELSE
        Let Nm in Cand be s.t. COUP(Gi*, {Nj.G | N in N.succ – Nm}) is minimal
        OBC(Nm, V)
  Incorporate(N, V) // updates N.G
}
```

Figure 5.13: Algorithm of OBC (Order-Based Clustering)

C-COI (Conceptual Clustering for Object Identification) applies an iterative optimization process to solve the order dependency feature of OBC-M. It reorders the variable clusters based on the latest result of OBC-M. The process recursively extracts a list of variables from clusters, and then, interleaves the lists. It first takes out an element from the largest cluster, then picks another element from the second largest cluster, and so on. Consequently, by placing variables from different clusters one after the other, this procedure returns a measure-dependent dissimilarity ordering. Biswas's work

demonstrates that interleaved orders produce better clustering trees and stable final groupings [Bis94]. A coupling-based criterion is used to measure the alternative solutions obtained by subsequent executions of the OBC-M:

$$SUM-COUP(G) = \sum_i COUP(G_i) \tag{5.6}$$

When applied to an entire hierarchy, the calculation of this criteria is limited to its target level grouping. Figure 5.14 briefly describes the algorithm of C-COI:

Given $\{V_{initial}\}$, a list of global variables (random order)
Apply OBC-M($\{V_{initial}\}$) to build $\{C_{initial}\}$, an initial tree

REPEAT
Reordering($\{C_{initial}\}$) to get $\{V_{new}\}$, the list of variables in a new order
Apply OBC-M($\{V_{new}\}$) to build $\{C_{new}\}$, a new tree
IF (SUM-COUP($\{C_{initial}\}$) < SUM-COUP ($\{C_{new}\}$)) *THEN*
keep $\{C_{new}\}$ as an initial clustering tree, *CONTINUE*
ELSE
keep $\{C_{initial}\}$ as final clustering tree, *STOP REPEAT*

Figure 5.14: Algorithm of C-COI (Conceptual Clustering for Object Identification)

Now we apply our C-COI (Conceptual Clustering for Object Identification) to the example introduced in Figure 5.10. The result is illustrated in Figure 5.15. The parameters are set as: MIN_COH = 65% and TREE_HEIGHT = 4.

In the beginning, the variables are presented in the random order as: ***a-b-c-d-e-f-g***. We first create the initial tree with a single node which contains the variable ***a***, or simplified as node ***a*** (see Figure 5.15(1)). When variable ***b*** is treated, the algorithm generates a new node to hold ***b***, and then it creates a new parent node to contain nodes ***a*** and ***b*** (see Figure 5.15(2)). The node creation pattern repeats when variables ***c***, ***d***, and ***e*** are added. However, after variable ***f*** arrives, the flat hierarchy structure is broken: node ***f*** has much higher cohesion value with node ***e*** than with the rest of the cluster. Therefore, this variable node forms a separate cluster with node ***e*** (see Figure 5.15(4)). When variable ***g*** is added, it shows that node ***g*** has been proved to be even more "similar" to ***f*** than ***e***. As a

result, nodes ***g*** and ***f*** form another highly cohesive fourth-level cluster (see Figure 5.15(5)).

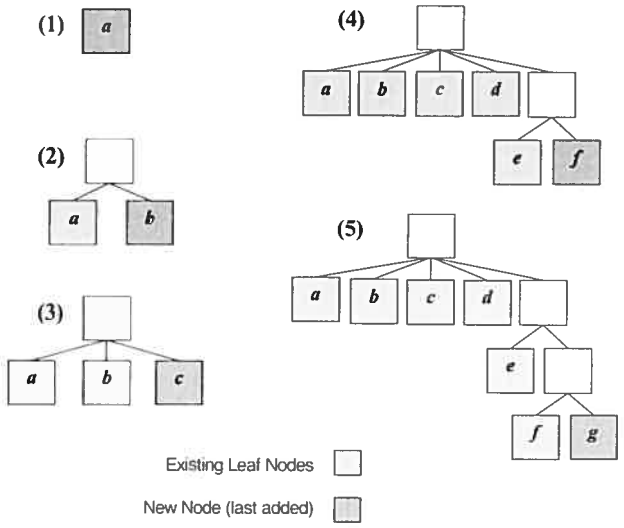


Figure 5.15: Conceptual Clustering Results in Different Presentation Ordering

5.3 Dynamic Featuring Technique

In the previous section, we have introduced two static featuring techniques to discover object models inside static source code. We have also developed a dynamic featuring technique to extract object models based on the observation of legacy system's execution.

The routines and global variables in the source code can be viewed as source code entities, or simplified as "Entity". During the execution of the legacy system, the entity invocation/reference information is captured. We use our dynamic analysis tool "Dynamic-Analyzer" to generate the dynamic entity interaction diagram (see example illustrated in Figure 5.16). The entity is represented as nodes in the interaction diagram. The construction rules are defined as following:

- Entrance entities (main routines) lie on the first layer.
- If one routine (E_1) uses or modifies the global variables (E_2), a link will connect E_1 and E_2; E_2 will lie at one-level layer lower than that of E_1.

- If one routine (E_1) calls another routine (E_2), a link will connect E_1 and E_2; the receiver entity E_2 will lie at one-level layer lower than that of the caller entity E_1.

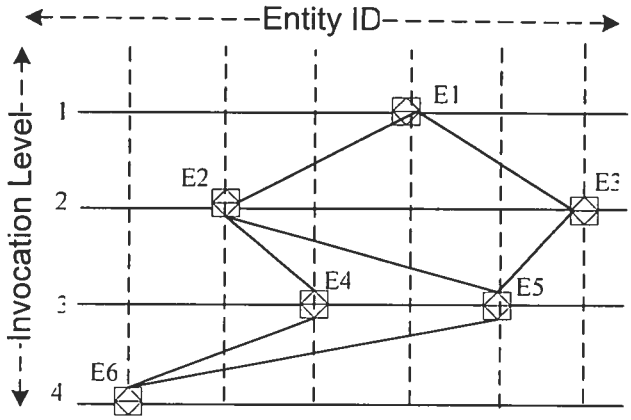


Figure 5.16: Sample of Dynamic Entity Interaction Diagram

The linkage shows the degree of coupling between two entities. We use the algorithm showed in Table 2 to calculate the coupling metrics among any two entities that appear in the interaction diagram.

1. Suppose the total invocation level is M. For illustration purpose, here we assume it is 10.
2. Arbitrarily we assign each pair of entities 0 out of M (10) degree of coupling values.
3. Further in the diagram, for each pair of entities, if they have direct link (level difference is 1), then it has as high as 9 (Total_Level - diffrence=10 - 1=9) degree of coupling value for this pair of entities.
4. Accordingly, for each pair of entities, if they have linkage expanded to two levels (level difference is 2), such as E1->E2->E5 in Figure 5.16, then {E1,E5} pair has a (Total_Level – difference =10 - 2=8) degree of coupling value.
5. Repeat step 4, until linkage expand to Total_Level -1 levels (level difference is 9).
6. Calculate the sum of coupling values for each entity pair.
7. Merge entity pairs into different sets according to their coupling values: if a pair's coupling value is larger than a threshold, then merge the entities of this pair into one group.

8. Continue at step 7, until reach an end.

```
Set_up (Coupling Threshold):

For (i=1 to Entity_Number)
        For (_=1 to Entity_Number)
                For (difference =1 to Total_call_Level -1)
                        IF (i!=_)  THEN  Coupling_difference (Ei. Ej)= 0;
//Arbitrary assign each pair of entities in all the levels to have 0 degree of coupling values.

For (difference =1 to Total_Call_Level -1)
        For (Each Pair(Ei. Ej) in graph )
                IF  ( (Distance (Ei.Ej)=difference) &&  Link (Ei.Ej))  THEN
                        Coupling_difference(Ei.Ej)+=Total_Level-diffrence;
//Calculate the coupling value for each pair of entities with any distances, the shorter
distance //they have, the higher coupling value these pair of entities will possess.

For (i=1 to Entity_Number)
        For (_=1 to Entity_Number)
                For (k=0 to Total_Level – 1)
                        IF (i<>_ &&  Coupling_k(Ei.Ej)>0) THEN
                                Coupling(Ei. Ej)+=Coupling_k(Ei.Ej);
//Get the final coupling value for each entity pair

Group entity sets according to Coupling (set [Entities])_values> Threshold :
Repeat
    If
        Coupling (set [Ei. Ej])_values> Threshold:
        Coupling (set [Ej. Ek])_values> Threshold:
    Then
        Group (Ei. Ej. Ek) as  one set:
        Coupling ( set [Ei. Ej. Ek]) = Min (Coupling (Ei. Ej), Coupling (Ej. Ek)) *0.8
Until Coupling Value for Each set<= Threshold:
```

Table 5.2: Dynamic Object Identification Algorithm

By this way, we will obtain a divided code entity group set. Each group has a number of different entities, among which have reasonably high degrees of coupling values. Performing this algorithm will generate a set of high cohesion code modules in a legacy system, thus to facilitate the discovery of class models in procedural code. The maximum invocation level will be decided at the run time depending on the real execution results of dynamic entity interaction diagram.

The real difficulty that we met in the experimental practice is the overwhelmingly large amount of entities that are involved in the diagram. In many cases, it becomes a disturbance to distinguish between different entities. To avoid such inconvenience in visualizing dynamic entity interaction diagram, we use two methods to solve this problem. One is to only visualize a limited amount of system functionalities at a time. This will limit the amount of source code modules involved on the visualization screen. The other one is to use the stored data from database to calculate the coupling values of each entity pairs, rather than visualize them on the screen. By this way, we are able to analyze the whole entity interaction space which is generated from the dynamic information captured during program execution period.

5.4 Summary

In this chapter, we have discussed the object-oriented re-architecturing of legacy systems. The focus is on the object-oriented modeling of each migration unit (decomposition part) of the system. To solve the object identification problem, we have presented three techniques that we have developed, namely the rule-based class recovery, static featuring technique, and dynamic featuring technique. For the static featuring technique, we have presented two object identification algorithms which are based on a genetic algorithm and a conceptual clustering algorithm. We note that, in a real-world practice, one should not apply only some of these techniques to solicit an object model from a legacy system. Rather, we should consider using all of them, and compare, analyze, evaluate the results with human intervention, and eventually optimize those results to obtain an optimum solution.

Chapter VI. Progressive Migration

Legacy systems are often large multi-year systems that preserve a great value to their owners. Migrating such systems in a single deployment process will create a high risk of failure, and the process will be time consuming, which also means high costs. To avoid the disadvantages of a one-shot migration, we have developed an incremental migration approach to solve this problem. Firstly, we analyze the target legacy system by using various techniques. Secondly, according to different decomposition strategies, we break the system into logically cohesive partitions. Various system decomposition outcomes will further be analyzed, compared, and optimized with human interventions, and synthesized to generate an optimum solution. The final system partition will eventually produce a set of migration units (MUs). Thirdly, the migration process is designed to contain multiple iterations, each of which will only focus on migrating a certain part of the target system (in our case, the MU), leading to an increase in the portion of the migrated code and to a respective decrease of the legacy code. In the end, the target legacy system will be progressively migrated into the OO paradigm.

6.1 Incremental Model

According to the system decomposition, source code modules will be clustered into groups that can be migrated by a single team within reasonable time limits. Such kind of a partition of target system is therefore defined as a migration unit (MU), which can be dealt with as a fundamental element in the progressive migration process. In Chapter 5 we have discussed the generation of object models from a legacy system. The re-architecturing work hereby will produce an object model for each MU. The

implementation of a certain MU is thus conducted by generating target OO language source code based on the object model of that MU.

To successfully implement all the object models into target OO language, we have designed the incremental migration model (see Figure 6.1) to fulfill the progressive re-engineering strategy.

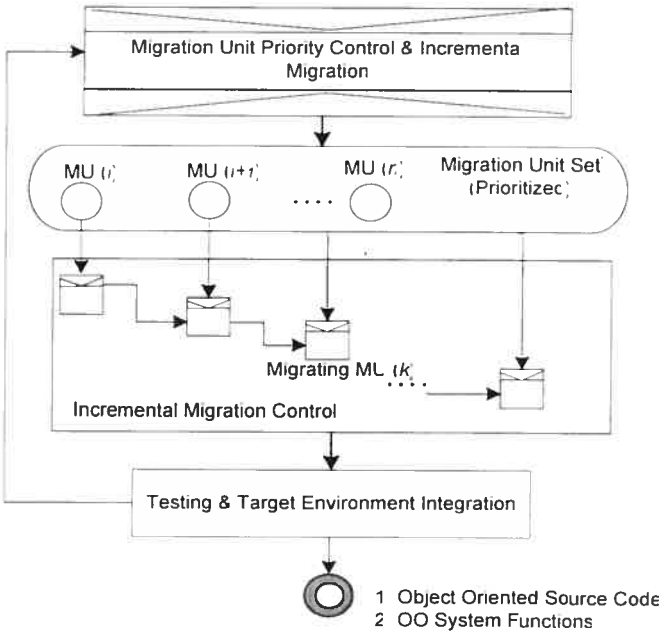


Figure 6.1: Incremental Migration Model

We first prioritize the MUs based on several criteria. Then, we select one MU with the highest priority, and migrate it into the target OO platform according to its object model. During the next iteration, since the situation has changed, we need to re-evaluate the priority order which was produced in the previous stage. Consequently the incremental migration control will re-prioritize the MUs, and find out the next MU to migrate. Such repeatable pattern will continue until all the MUs are successfully migrated.

The implemented MU will first pass the unit testing to evaluate the correctness. Later, when a new MU has been migrated, an integration testing will be conducted. Increasingly, the whole legacy system will be migrated into object-oriented platform by progressively implementing the system into the OO paradigm (see Figure 6.2). Since the

modernized code cannot communicate with the remaining of the legacy system, a bridge code will be created to help the two parts cooperate to test the correctness of the migrated parts. Although our approach constitutes an in-operational migration process that allows the target system to run with no downtime, we mainly use bridge code for the purpose of the unit/integration testing and validation of the migrated MU.

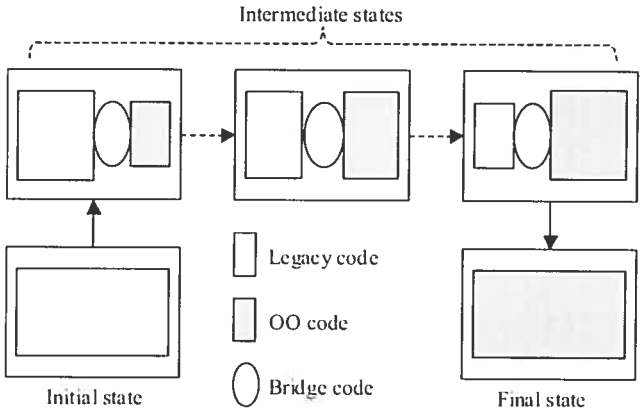


Figure 6.2: Incremental Migration Illustration

The construction of the bridge code is based on the target migration OO language. In our experimental study, we use Java as the implementation language. Therefore we can use many desirable features/techniques that provide by the target OO platform to construct our bridge code. The detail will be presented in the later section.

Benefits of Incremental Migration

Firstly, such an approach substantially reduces the risk and the cost of a migration project. In fact, since no single deployment strategy is adopted, one needs not to allocate all the resources to the migration project at once. In a relatively short period, a part of the legacy system will be renovated and available for a (partial) evaluation of the migration impact. Secondly, at the starting point, preference will be given to the most tractable parts of the legacy system, while the remaining parts will be considered with respect to their dependency (service requests) on the previously migrated parts. Thus, a reworking of the code that has already been migrated will be avoided. Finally, our approach provides a tangible management mechanism by dividing the whole project into separate tasks (to be

assigned to possibly independent teams) which will be more flexible to assign and manage.

6.2 Migration Sequence Prioritization

Since legacy systems are normally large, progressive migration can greatly reduce the reengineering complexity and risk. Moreover, it requires a relatively small amount of resources at a single point of time. However, after successfully decomposing legacy system, we will face the question of in what sequence these MUs should be migrated. Our approach is to first analyze the legacy system characteristics, then evaluate the available reengineering resources, finally adopt a suitable priority strategy, apply it on all MUs, and select the best candidate to migrate first. After the implementation of one MU, the prioritization procedure will be performed again to find out the next most suitable MU, and iterate the migration process until all MUs are successfully migrated.The problem then becomes what kind of criteria should we adopt to rate the migration priority for each MU? We have selected the following criteria to reflect the MU priority value:

1. *Module complexity:* Several metrics are applied to determine the complexity degree of each MU. We primarily use three metrics, namely Cyclomatic Complexity, Module Complexity (coupling via parameters and global variables), and Structure Complexity (maximum depth of reference structure chart).
2. *Isolation degree:* Relationships with other MUs are calculated in the form of module reference times (weights). It is calculated as the ratio of the function calls from the source code modules in one MU to those modules in other MUs. The lesser degree of dependence with the rest of the system, the higher isolation degree it will have.
3. *Module importance*: With the help of dynamic behavior analysis tools, those MUs that perform more important tasks in the system will be ranked at a higher importance.
4. *Understandability*: A subjective analysis combined with complexity metrics are used to indicate the understandability degree of a particular MU. Based on testing on a number of randomly selected sample code segments, testers will be asked to provide a

subjective assessment of the legacy program understandability. Meanwhile, mainly three metrics, namely Cyclomatic Complexity, Comment Density, and Name Uniqueness Ratio are used to evaluate the understandability of modules in a certain MU.

5. *Environment independency*: If a given MU does not rely too much on a specific application environment (such as OS, hardware, implementation language, etc.), it will have higher degree of environment independency. For example, if a MU has a large percentage of code which uses a specific language-related API to implement the GUI part, and also if the target OO system uses a language that does not support that specific API, then, most of the legacy code which uses the special API in that MU will be changed in the migrated system. This will bring more modifications into the implementation. Therefore, the environment independency degree of a MU reflects the extra effort it needs to migrate that MU. Since if a MU is not depend on a specific programming environment very much, that will mean that most of the code segments will not need a great amount of changes when migrating to target OO language. Hence it reduces the implementation complexity during the migration process.

The above five criteria will be applied to each MU, and their combination will produce the migration priority sequence. However, there are many subtle issues related with the calculation of combining two criteria. One concern is that their measurements may not be coincident with each other, thus making it almost impossible to apply a same synthesis mechanism to generate the joint result. For example, when we consider the criteria (1) *module complexity* and (2) *isolation degree*, the caliber rang for (1) could be $[0, \infty)$, while that of (2) could be [0, 100%]. If we simply use Boolean algebra to calculate the result by summarizing them, then, the influence of criterion (1) will be much larger than that of criterion (2). In fact, a small portion of the difference in (1) will quickly overwhelm the weight of (2).

To solve this problem, we apply a fuzzy expert system [Leo98] to synthesize the results of those criteria, and calculate the final MU sequence priority order. The fuzzy expert system uses fuzzy logic instead of boolean logic to solve the problem. In another word, a fuzzy expert system is a collection of membership functions and rules that are

used to reason about data. Unlike conventional expert systems, which are mainly symbolic reasoning engines, fuzzy expert systems are oriented towards numerical processing. The rules in a fuzzy expert system are usually similar to the following:

If *x is low and y is high* ***Then*** *z = medium*

Where *X* and *Y* are input variables, *Z* is an output variable, concepts low/high/medium are membership functions defined on *X, Y* and *Z* respectively. The part of the rule between the "if" and "then" is the rule's premise which describes to what degree the rule is applicable. The part of the rule following the "then" is the rule's conclusion that assigns a membership function to each output variable.

. For example, for the criteria "*module importance*", suppose the highest importance metric value is set as 10, and lowest is 0, we can define the member function as illustrated in Figure 6.3:

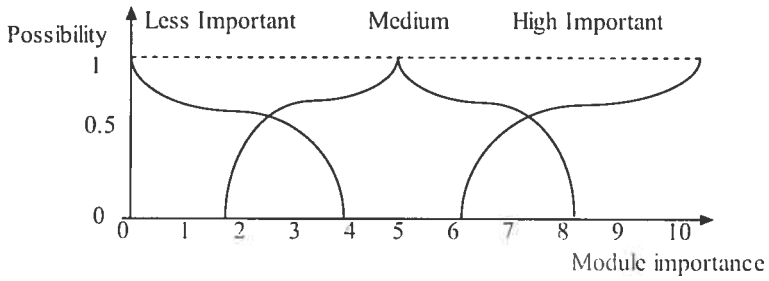


Figure 6.3: Member Function Definition

The membership function shows that, if the *module importance* metric value ranges in [0,4], then it is less important with possibility of range [100%, 0%]; if the metric value ranges in [2,8], then it is medium important with possibility of range [100%, 0%]; if the metric value ranges in [6,10], then it is very important with possibility of range [100%, 0%]. For any metric value varies from 0 to 10, we can find the possibility value of less, medium, and very importance. Hence, we can design a membership function for each criterion that we have previously defined. The metric value of each criterion will be fuzzified with its membership function. During the inference process, the truth value for the premise of each rule is used to generate the conclusion, and all the rules will be

applied to obtain a priority by applying fuzzy synthesis in the fuzzy expert system. Therefore, in the end, we will be able to combine the results of all these criteria, and calculate the MU sequence priority order.

Here we give an example to illustrate how to apply fuzzy expert system to analyze with only two criteria—*model complexity* and *isolation degree* to produce the combined migration priority value of a specific MU (see Figure 6.4).

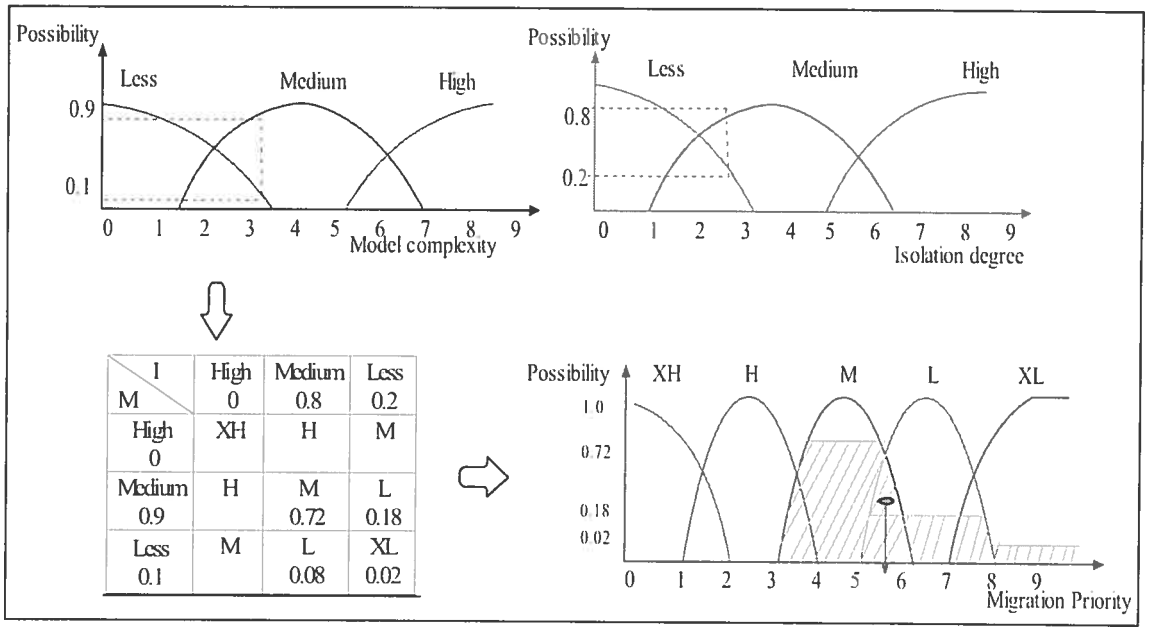


Figure 6.4: Fuzzy Synthesis of Migration Priority by Two Criteria

We have variables:

X is *module complexity*;

Y is *isolation degree*;

Z is priority value;

Membership functions for criteria *module complexity* and *isolation degree* are illustrated on the top left side and right side in Figure 6.4, respectively. Their metric values are used to find out the possibility values according to their membership functions. Here we have:

- For criterion *module complexity*, it has 90% possibility to be "medium" complex and has 10% chance to be viewed as "less" complex;
- For criterion *isolation degree,* it has 80% possibility to be "medium" isolated and has 20% of chance to be viewed as "less" isolated;

The rules defined in this fuzzy expert system are detailed as follows:

1. **If** *X* is high (H) with possibility of *a* and *Y* is high (H) with possibility of *b*,
   then Z = extra high (XH) with possibility of *a*b*;
2. **If** *X* is high (H) with possibility of *a* and *Y* is medium (M) with possibility of *b*,
   then Z = high (H) with possibility of *a*b*;
3. **If** *X* is high (H) with possibility of *a* and *Y* is less (L) with possibility of *b*,
   then Z = medium (M) with possibility of *a*b*;
4. **If** *X* is medium (M) with possibility of *a* and *Y* is high (H) with possibility of *b*,
   then Z = high (H) with possibility of *a*b*;
5. **If** *X* is medium (M) with possibility of *a* and *Y* is medium (M) with possibility of *b*, **then** Z = medium (M) with possibility of *a*b*;
6. **If** *X* is medium (M) with possibility of *a* and *Y* is less (L) with possibility of *b*,
   then Z = less (L) with possibility of *a*b*;
7. **If** *X* is less (L) with possibility of *a* and *Y* is high (H) with possibility of *b*,
   then Z = medium (M) with possibility of *a*b*;
8. **If** *X* is less (L) with possibility of *a* and *Y* is medium (M) with possibility of *b*,
   then Z = less (L) with possibility of *a*b*;
9. **If** *X* is less (L) with possibility of *a* and *Y* is less (L) with possibility of *b*,
   then Z = extra low (XL) with possibility of *a*b*.

The rules that we have used for these two criteria are rules (5), (6), (8), and (9). The fuzzy synthesis calculation is illustrated by the matrix showed in Figure 6.4 on the left-bottom side. The intermediate results are calculated as volume of each membership function, illustrated in Figure 6.4 on the right-bottom side. The last step is to de-fuzzify the intermediate results by calculating the gravity centre of the total volume, and generate the priority value for that MU.

6.3 Bridge Code

During the migration of a certain MU, we will face a difficult question that the renovated part with OO code needs to call the functions provided by the rest of the system which is written in legacy programming language, and vice versa. Moreover, during the phase of units and integrations testing of migrated MUs, we further need to evaluate the functional equivalence of implemented parts, which also means the requirement of inter-communication between these two parts. However, by nature, these two parts, one written in OO programming language, the other written in legacy programming language, can not communicate with each other.

To solve this problem, bridge code has to be created to help these two parts cooperate with each other. In our experimental practice, we mainly use the features and techniques provided by the target migration OO language/platform to construct the bridge code. Since we primarily use Java as the destination OO language, we are able to apply a broad range of techniques that supported by Java to facilitate our work. The Java Native Interface (JNI) is the primary technology that we applied to build up the connections between the migrated OO part and the rest of the legacy program. JNI is the native programming interface for Java which is part of the JDK. The JNI allows Java code that runs within a Java Native Interface to operate with applications and libraries written in other languages, such as Fortran, Ada, Pascal, C, Cobol, assembly, etc. Meanwhile, it also allows those native languages to call the Java methods.

For example, in our case studies, we have migrated legacy systems written in C into OO systems written in Java. During the incremental migration phase, the hybrid systems include two major parts: the C side legacy code part and the Java side migrated OO code part. The communication between these two parts is conducted by the bridge code implemented in JNI, as illustrated in Figure 6.5.

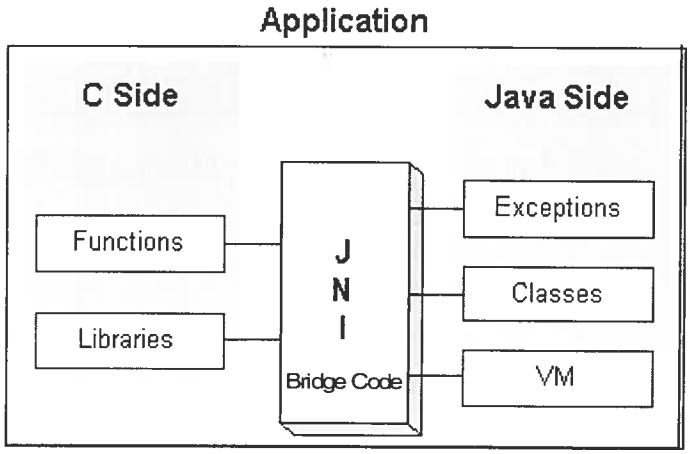


Figure 6.5: Bridge Code Construction with JNI

The functions and libraries that implemented in legacy system can be accessed by the classes defined in the Java program side. Figure 6.6 illustrates the calling of native legacy language functions from migrated Java code side. This diagram shows the utilization of JNI to construct bridge code to access the legacy program parts from the migrated Java program parts. It includes calling legacy libraries/API, legacy routines, accessing native (legacy language) debugger, exception handler and native run time type checker, etc. It is easy to see that the JNI serves as the glue between the migrated Java code and the rest of legacy system code.

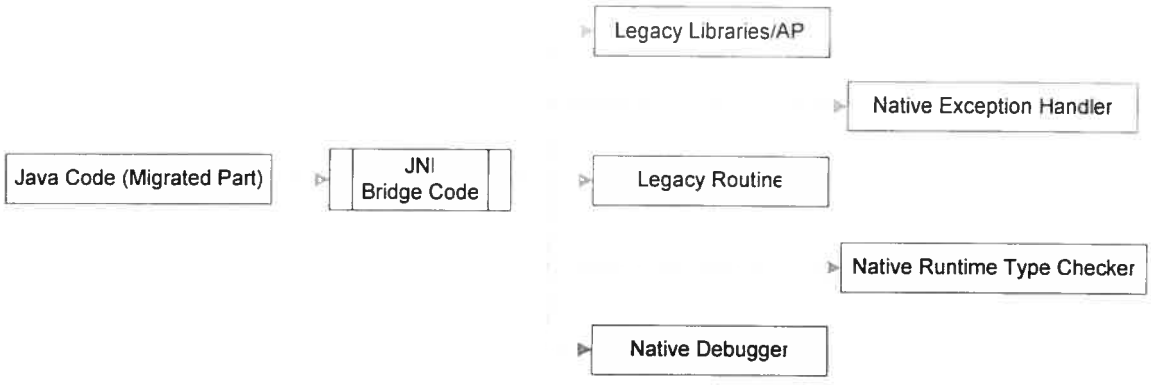


Figure 6.6: Java Code (Migrated Part) Calls Legacy Routines

Meanwhile, the classes, virtual machine, and exceptions are all accessible to the routines defined in the legacy program side (see Figure 6.7). The JNI framework allows the native methods (routines in legacy system) to utilize Java objects in the same way that Java code uses these objects. A native method can create Java objects, and then inspect and use these objects to perform its tasks. A native method can also update Java objects that it created or that were passed to it, and these updated objects are available to the Java application. Thus, both the native language side and the Java side of the hybrid legacy system can create, update, and access Java objects and then share these objects between them. Furthermore, native methods can easily call Java methods. In particular, one can catch and throw exceptions from the native methods and have these exceptions handled in the Java application. Figure 6.7 shows that the legacy program can use JNI to construct bridge code to access Java libraries, Java Virtual Machine, exceptions, Java classes, call Java methods, etc.

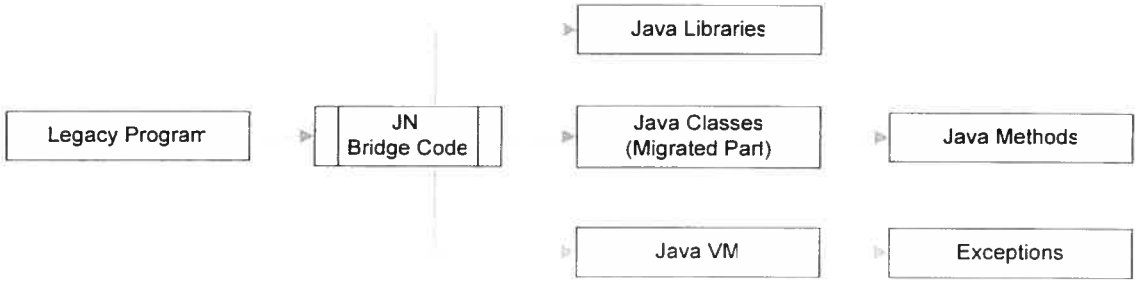


Figure 6.7: Legacy Routine Calls Java Code

6.4 Summary

In this chapter, we have discussed our progressive migration approach and related issues. According to the decomposition strategies, the whole system is broken into logically cohesive partitions. Various system decomposition outcomes will further be analyzed, compared, and optimized with human interventions, and synthesized to generate an optimum solution. The final system partition will eventually produce a set of migration units (MUs). The incremental migration process is designed to contain multiple

iterations, each of which will only focus on migrating a certain part of the target system (in our case, the MU), leading to an increase in the portion of the renovated code and to a respective decrease of the legacy code. A fuzzy expert system is applied to prioritize the migration sequence. During the migration of a certain MU, we also face a difficult question that the renovated part in OO code needs to call the functions provided by the rest of the system which is written in legacy programming language, and vice versa. To solve this problem, a bridge code has to be created to help these two parts cooperate with each other. We have discussed how to apply Java Native Interface (JNI) to build the connections between these two parts.

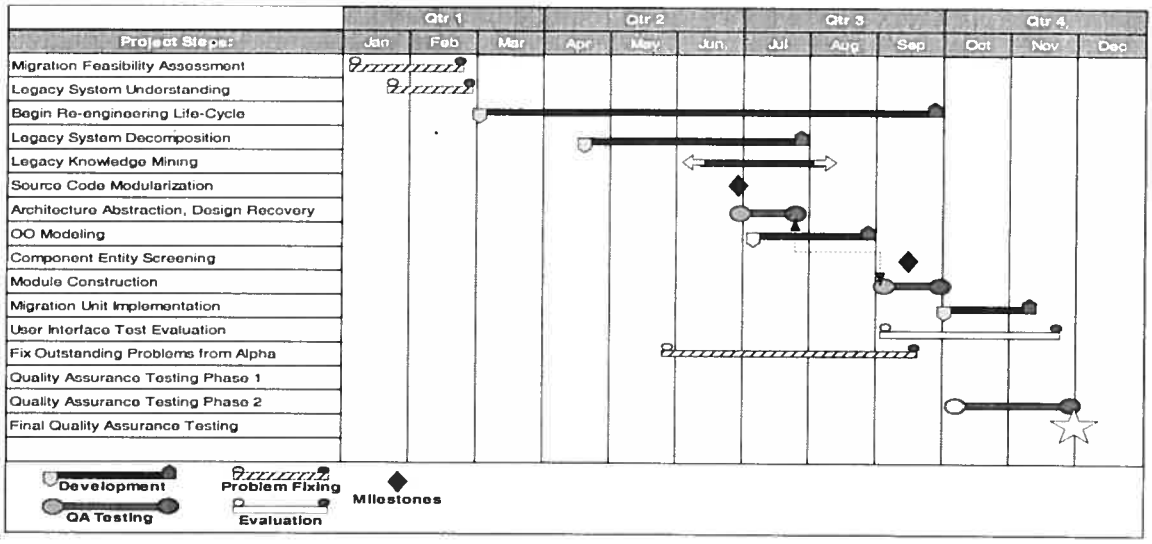
Chapter VII. Migration Project Supporting System

With the rapid progress of internet technology, more and more software projects adopt an e-development to facilitate the software development process in a world wide context. For a modern software migration project, one development team may not have all the expertise that is required for the project. Several teams may cooperate together to accomplish the migration work. Even within one team, team members may not be able to work at the same location. A distributive collaboration platform is highly desired to facilitate the migration development over the internet. However, collaborative legacy system migration activity itself is a complex orchestration. It involves many people working together without the barrier of time and space differences. The issues of efficient project planning, web-based task progress monitoring, and communication supporting become important concern in a migration project. In this chapter, we present our approach to tackle these three important issues. In our research, we have designed a prototype, *Caribou,* to contain our solutions. It also works as an integrated migration project supporting environment.

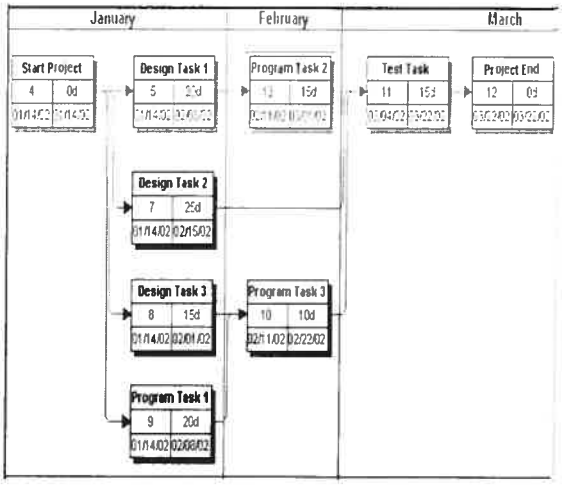
7.1 Facilitating Migration Project Planning/Scheduling

In a legacy migration project, planning/scheduling is very crucial to the project success. The project plan includes the schedule that shows who will do which task and when. A plan is like a vision of the project, which will help the project manager virtually

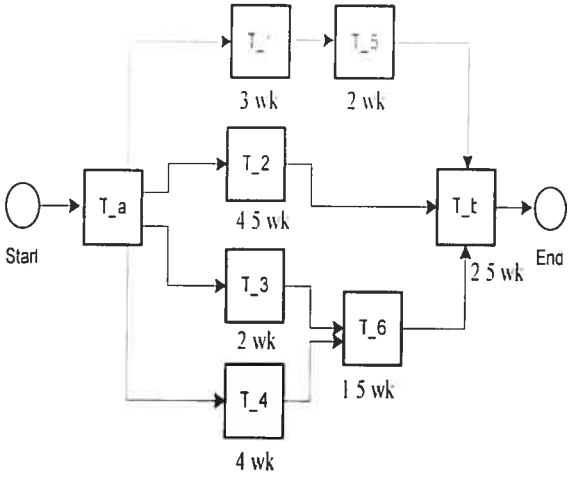
see (predict) the project as it would move through the various phases. In our Caribou project supporting system, we provide tools to support three major project planning/scheduling techniques, namely Gantt chart, PERT chart (the Program Evaluation and Review Technique), and Critic Path Method (CPM). Moreover, we have designed a planning method which applies dynamic task notation and workflow technology to construct the visual representation of project scheduling.



(i) Gantt chart



(ii) PERT chart



(iii) CPM diagram

Figure 7.1: Supporting Classical Project Scheduling Techniques in *Caribou*

A Gantt chart is a graphical representation of the duration of tasks against the progression of time. *Caribou* provides tools to support the standard Gantt chart drawing notation elements and the editing of the chart. Figure 7.1 (i) illustrates a sample project scheduling with a Gantt chart in *Caribou*.

The order in which tasks occur is an important part of project planning. A PERT chart displays the tasks in a project along with the dependencies between these tasks. CPM is a technique that analyzes what activities have the least amount of scheduling flexibility (i.e., are the most mission critical) and then predicts project duration schedule based on the activities that fall along the "critical path". Activities that lie along the critical path cannot be delayed without delaying the completion time for the entire project. Figures 7.1 (ii) and (iii) illustrate the project planning with PERT and CPM in *Caribou*.

However, for all these three traditional project planning/scheduling techniques, there is a lack of capability to model the synchronization semantics of the dependency of different tasks. For example, for a task dependency relationship, when Task_a finishes, it will lead to the start of three other tasks, namely Task_1, 2, and 3. This can be represented by PERT/CPM as illustrated in Figure 7.2:

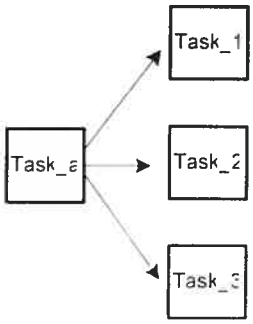


Figure 7.2: Classical Task Dependency Diagram

But they cannot express many other semantics, such as: when Task_a finishes, it only leads to the start of one of these three tasks (Task_1, 2, and 3); or when Task_a finishes, it may leads to the start of one or more than one of these three tasks.

To solve this problem, we adopt the workflow technology to build our project planning visual modeling solution. We use the modified version of the dynamic behavior modeling

notation from YAWL (Yet Another Workflow Language) [Aal03] to construct the visual representation of project planning activity elements (see Figure 7.3). The semantics of the notions are defined in Table 7.1.

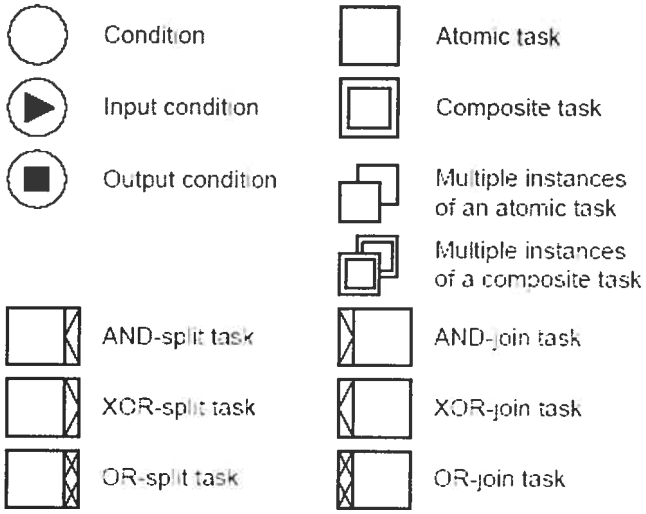
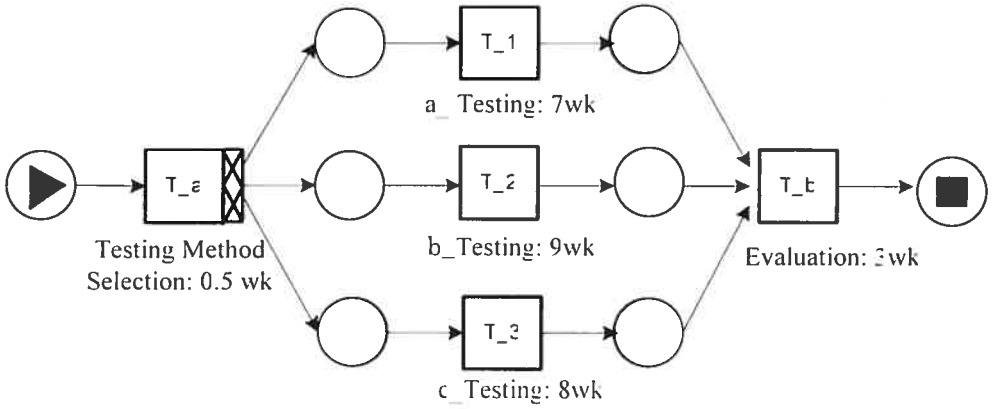


Figure 7.3: Planning Notation in *Caribou*

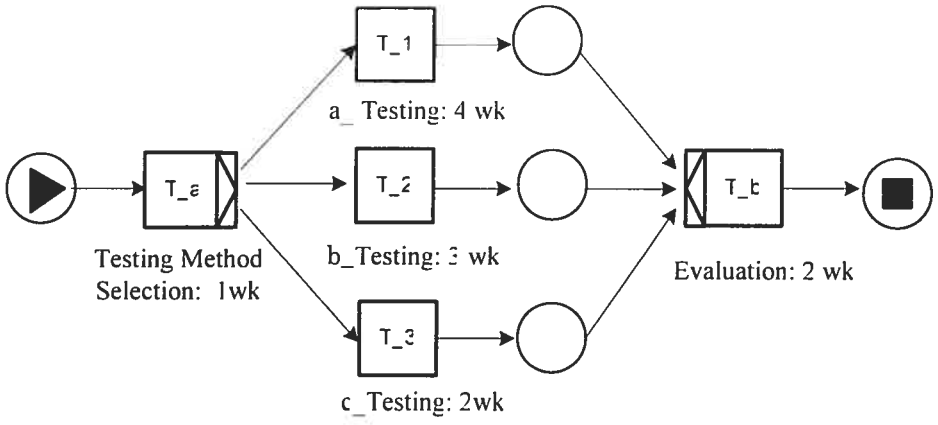
Notion Name	Semantics of the Notion
Condition	A certain circumstance of a specific state
Atomic task	The smallest task unit
Composite task	A task that includes other smaller sub-tasks
Input condition	The start point of a project
Output condition	The end point of a project
Multiple instances of an atomic task	More than one invocation of a specific atomic task
Multiple instances of a composite task	More than one invocation of a particular composite task
And-split task	When a task finishes, it generates several conditions
XOR-split task	When a task finishes, it generates only one of those conditions

OR-split task	When a task finishes, it generates at least one of those conditions
And-join task	When all of the conditions are satisfied, this task will be performed
XOR-join task	When only one of the conditions is satisfied, this task will be performed
OR-join task	When at least one of the conditions is satisfied, this task will be performed

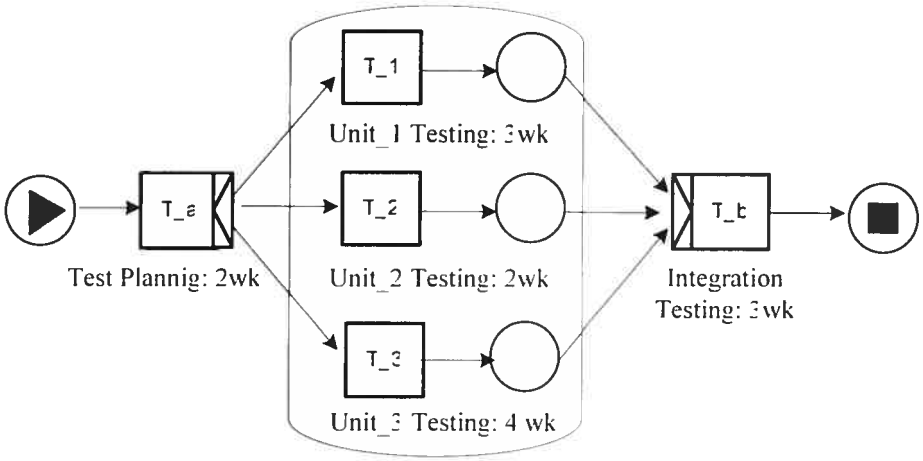
Table 7.1: Semantics of the Modeling Elements



(i)Task "Evaluation" executes when one of the three preceding tasks completes. Depending on how many testing methods "T_a" has selected, "Evaluation" performs at least once, and at most three times.



(ii)Task T_a only generates one subsequent task. Task "Evaluation" is executed only once, when one testing is completed.



(iii)Task "Integration Testing" is executed only once, when all preceding tasks have completed.

Figure 7.4: *Caribou* Enhanced Synchronization Notation in Project Planning

Figure 7.4 illustrates an example of applying the visual notations to construct a test plan. We can see that this adapted visual representation can express a variety of project planning activity semantics.

We have developed the project planning workflow module in *Caribou*. It includes the following two aspects:

- Designing and implementing an embedded workflow management system (WfMS) within *Caribou*. The workflow server later supplies the web-based, collaborative project task monitoring.
- Modeling project tasks using rich semantic definition notation to specify their missions and synchronizing/procedural constraints. It later supports dynamical modification and adjustment of web-based collaborative tasks.

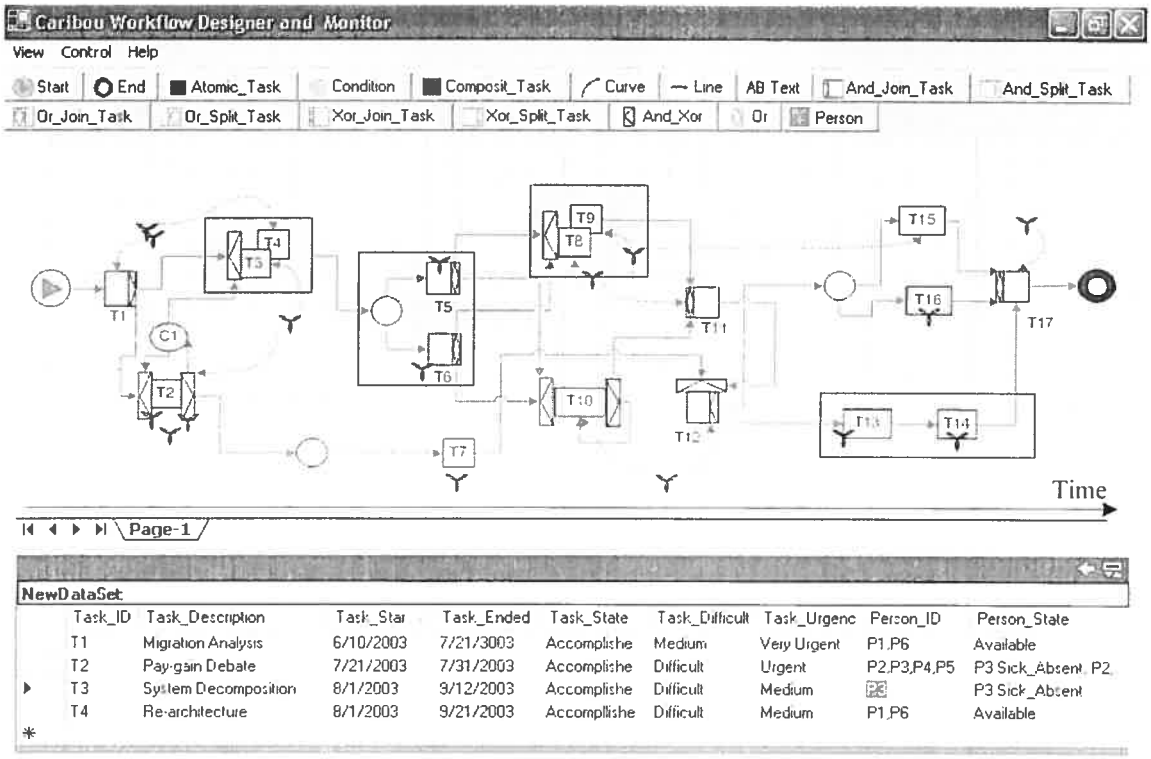


Figure 7.5: Project Planning in *Caribou*

Figure 7.5 illustrates the web-based workflow modeling of project planning in *Caribou*. The notation ♈ represents personnel. We can see that several people may work together on a same task; and one person can also participate in more than one tasks. The enhanced dynamic behavior modeling notation is applied here in project task planning.

7.2 Monitoring Project Task Progress

Collaboration status monitoring and controlling is one of the most important issues in a software development project. A dynamic supervision of ongoing tasks has to be deployed. It will regulate both collaborative group members and external management, thus to ensure the development project to be in schedule. Moreover, it also should adopt some proper actions in case of schedule slippage. In this section, we present our approach in fulfilling this objective.

7.2.1 Modeling Collaborative Task Progress

Like any other kind of engineering practices, software development is a progressive process. There is a time line to distinguish different stages of achievement. It helps practitioners to track the historical performance as well as monitor and control the activities that are presently undergoing. Unfortunately, for a web-based collaborative legacy migration project, due to the distributed and dynamic nature of web collaboration environment, this time line is not easy to define. Web collaboration has made all the participants in a virtual development venue. All these may require practitioners to have an efficient way to monitor and control their widely dispersed development activities.

To solve this complex problem, we start by defining the task progress model. Furthermore, we apply autonomous agent to collect quantitative data to measure the progress metrics in the legacy migration project. In addition, we also have defined a set of control measurements corresponding to different progress deviations.

In a legacy migration project, the collaboration consists of many tasks that involve different individuals. To monitor the whole collaboration, we should have a clear view of each task's progress status. Task generally can be viewed as an elemental work unit - a conceptually whole that is performed by one or several persons. It normally includes concrete detailed purposes, performance steps, and final results. We should note that a task is a logical definition and the number of participants can be more than one. The challenge lies in that, at any given moment, how can we measure a task's progress?

Our approach is to first define the task progress model which includes the progress metrics; then we utilize agents to dynamically collect task attribute information which reflects the progress metrics; in the end, we visualize them to reflect each task's progress status. To illustrate our approach, we first analyze task attributes. A collaborative software development task has the following two types of attributes: static and dynamic attributes (see table 7.2 and 7.3).

task name/code
task type
scheduled time
participant (number and names)
anticipated output result (description)
steps (name/code, description, number)
scheduled resource (name/code, type, quantity)

Table 7.2: Static Attributes

consumed time
consumed resources (name/code, type, quantity)
undergoing development function (name/code, number)
importance (quality requirement)
urgency (development time requirement)
stable output function (name/code, number)
unstable output function (name/code, number)
undergoing step (name/code)
emergent event (name/code, type, number)
extra resource request (name/code, type, quantity)
distance from final objective estimation (heuristic measurement)
health index (heuristic measurement)

Table 7.3: Dynamic Attributes

A task's dynamic attributes strongly reflect its progress character. We have also noticed that under certain conditions, some static attributes can be changed into dynamic attributes. For example, at a given time, when an emergency event occurs, the static

attributes of participant number may be changed into dynamic attributes. A visual model, showed in Figure 7.6, illustrates the collaborative project task progress model in *Caribou*.

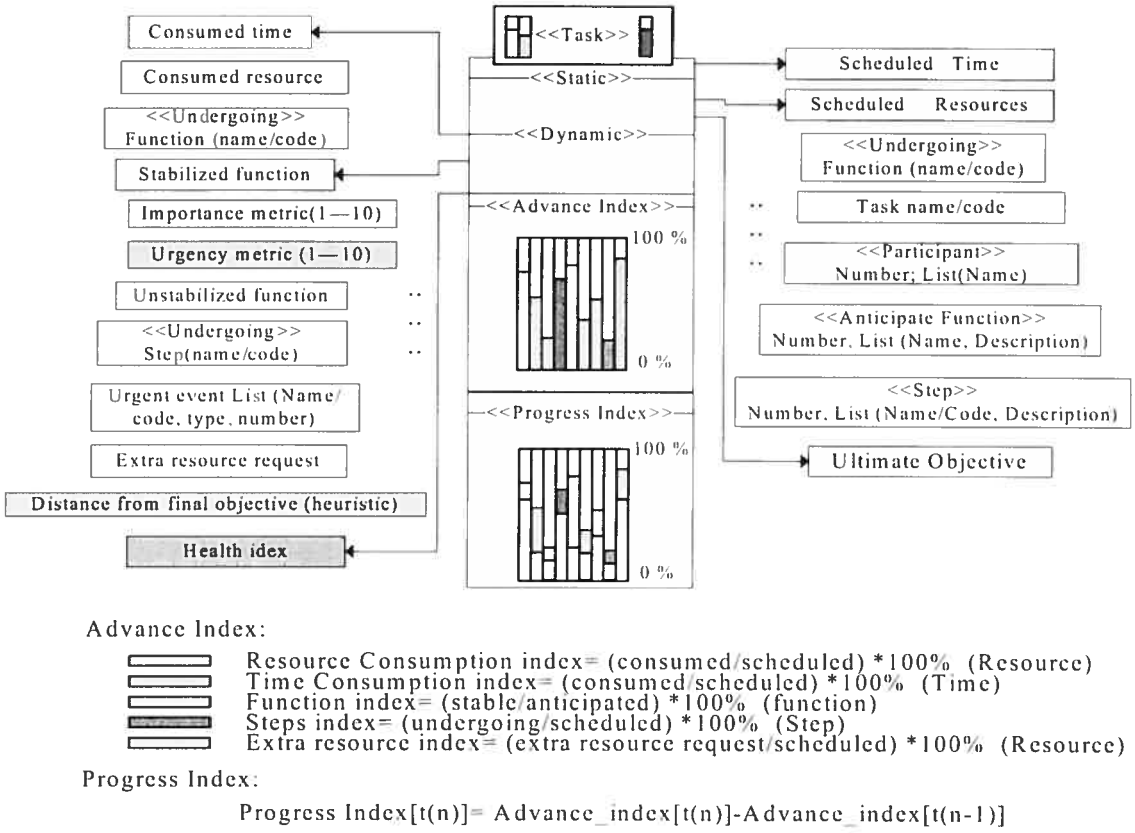


Figure 7.6: Caribou Collaboration Task Progress Model

It visually represents the degree of progress of each task of the whole migration project. At any given time, it shows the static (right) and dynamic (left) attributes of the observation task (center). In parallel, it shows the dynamic aspects in terms of advance and progress rate. The advance index (middle) indicates the status of this task; and the progress index (bottom) shows the relative increase during a period of time.

The status of a given task at any time is measured by a group of metrics, called "advance index". They together reflect the advance degree of the task. Here we give some of them, as illustrated in Table 7.4.

Resource_Consumption_Index= (consumed/scheduled) * 100%	(Resource)
Time_Consumption_Index= (consumed/scheduled) * 100%	(Time)
Function_Index= (stable/anticipated) * 100%	(Function)
Function_Quality_Index = (1 - unstable/anticipated) * 100%	(Function)
Steps Index= (undergoing/scheduled) * 100%	(Step)
Extra_Resource_Index=(extra resource request/scheduled) * 100%	(Resource)

Table 7.4: Progress Metrics Definition

Progress Index: The progress of a given task will be represented by visualizing the difference of "advance index" between the states during a period of time. They work together as a sign to show the increment of task status. Suppose the time interval between two given moments is $\Delta t=t2-t1$ where t1=start time, t2=end time. To measure a certain task, or a group of tasks, we assume the period of observation time Δt is same. Therefore, we define progress index for each metric as formula 7.1. The visualization of the calculation is illustrated in Figure 7.7.

$$Progress\_Index(t2)=Advance\_Index(t2)-Advance\_Index(t1) \quad (7.1)$$

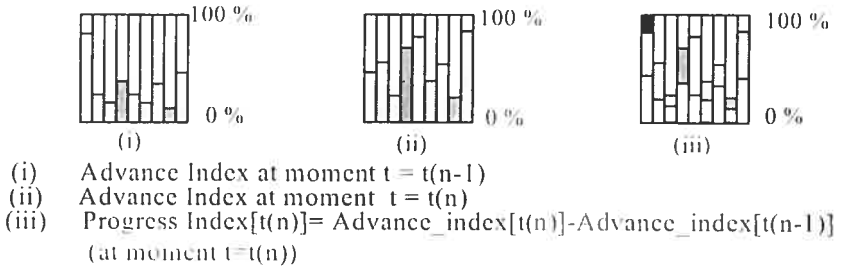


Figure 7.7: Calculation of Progress Index

7.2.2 Monitoring Project Tasks

After building up models for collaborative project tasks, our next objective is to monitor the current status of migration development activities. One crucial step to realize automatic monitoring is real-time activity data collection. Task agents is used to

automatically collect dynamic task attribute data and event information. In our prototype, we use graphic diagram to visualize the status of collaborative tasks and the project progress historical information. The task network has two dynamic aspects. One is the continuously generation and expiration of tasks; the other is the constantly changes inside existing tasks.

At a given time, the whole development may have a certain number and types of tasks that are undergoing. After a period of time, some tasks may have already been finished, while some new ones have been generated to suit for current project progress needs. Therefore, our whole collaboration network will regularly change its shape and organization. The comparison with project planning and scheduling will illustrate the variance between the plan and reality.

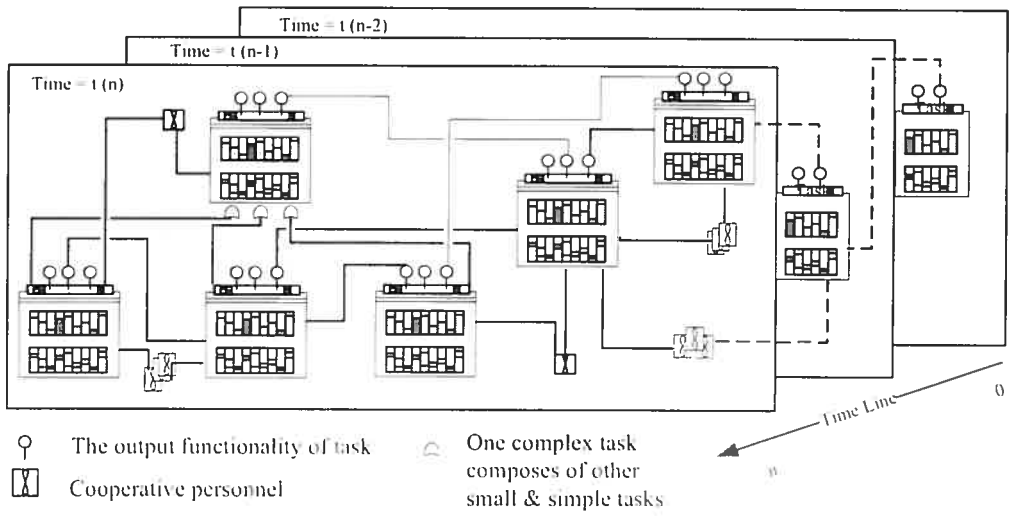


Figure 7.8: Collaboration Network

Each task has its own life cycle and may have many people cooperate within it; various events can affect its progress route. The dynamically collected activity data by task agent is used to generate the collaboration network diagram (see Figure 7.8). A task may use the outcome produced by others. We use a line to connect the requester task and the output result (represented by tiny circle) of a service provider task. Two tasks may "cooperate" with each other by providing services mutually. Each layer on the diagram represents the task network at a given time. To use the results provided by a historical

task that does not lie on the same layer of the active tasks, a dotted line will be used to connect two tasks between two layers. The same will also be applied on participant personnel. The lines between tasks thus represent the conceptual supporting relationships among them.

7.3 Supporting Collaborative Personnel Communication

In this section, we present the web-based migration project communication control module in *Caribou*. We first analyze and model human communication in a web-based development environment. This model will later help us to monitor and control participants' communication activities.

As we know, humans are the primary factor in a project. Many research results have shown that the communication between participants is one of the most efficient ways to conduct cooperation [Tea00][Nar00]. An important aspect of web-based communication is how individuals interact in a virtual group. A cognitive-based analysis has been used to evaluate interaction effectiveness [Mal01]. Further research emphasizes on the coordination of communication [[Pre00], and shows that participants' cooperation efficiency will largely affect the progress and quality of whole project.

7.3.1 Collaborative Software Development Communication Model

Meeting, discussion, pair programming, etc., are various collaboration forms. In a traditional environment, such activity is easy to obtain with oral language. While in a web-based development environment, this type of information exchange is not that convenient to acquire. In most cases, lack of an efficient way to monitor and measure communication activities remains one of the major challenges of networked collaboration [Pen02].

With the geographical separation and time difference, it is difficult to have a traditional meeting or discussion about a concerned issue. In many cases, people even may not be able to use on-line chat rooms to discuss, since their working hours may be

reversed because of time zone differences. People would rather use email or bulletin board to exchange their opinions [Hau01]. This type of communication raises another issue: how to quantitatively measure the quality of this sort of cooperation? How to reinforce the collaboration in terms of e-communication? The later one means, in e-development, if some people are sluggish in providing the demanded information required by others, how can we adopt more efficient actions to systematically avoid such situations? Furthermore, if we have an urgent question and do not know who is responsible or is potentially able to answer it, who should we discuss with? Moreover, after we have published such urgent questions on e-bulletin board, and have not gotten satisfactory answer, what should we do next? If the proper person simply does not have time to browse discussion board, even though we know that there must be someone who has the answer, yet we still cannot trigger her out. This could eventually sabotage the collaboration efforts.

Therefore, to solve the above problem, we have set up three major goals for our research prototype, they are: (i) to effectively convey the concerned messages to proper people; (ii) to secure the information solicitation mechanism; (iii) to measure the quality of collaboration by means of communication in a web-based e-development environment.

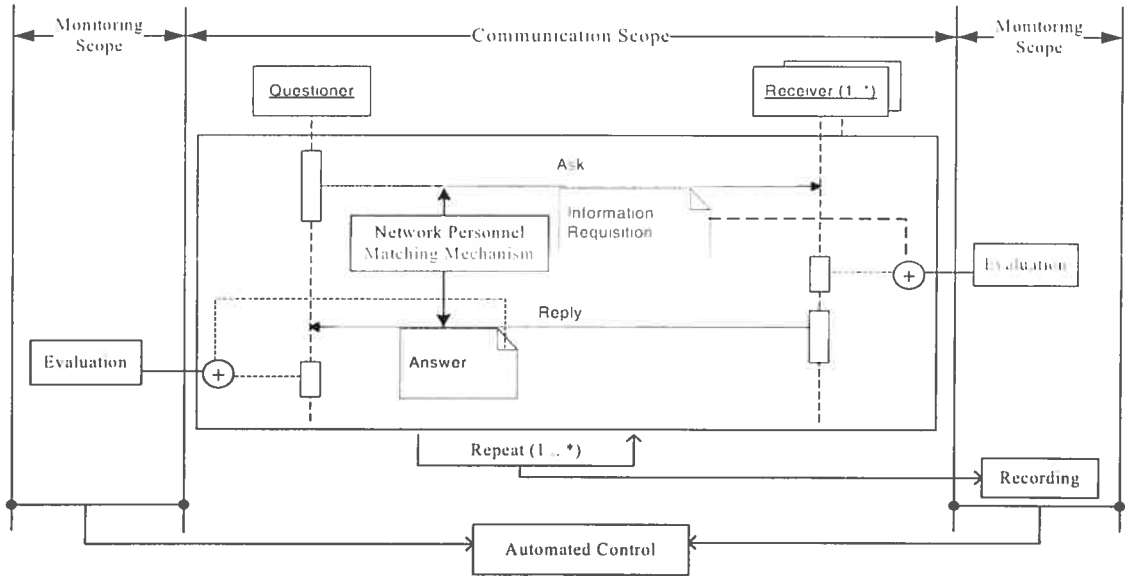


Figure 7.9: Collaborative Software Development Communication Model.

To realize our ultimate objectives, we first build up our collaborative software development communication model (see Figure 7.9).

There are two application scopes, namely communication scope and monitoring scope. The recording and evaluating of communication transactions are performed in monitoring scope. This means, those processes are purely related with performance measurement purpose. In contrast, the communication activities are fully executed within communication scope.

7.3.2 e-Development Communication in *Caribou*

To observe well human communication activities inside of web-based software development projects, we have built up the collaborative communication model in prior section. Now we apply this model to automate the monitoring and controlling measurements of personal communication activities in a collaborative e-development environment.

e-Development Personnel Matching Mechanism.

Here we provide our solution to the first goal in our prototype, which is to effectively convey concerned messages to proper people. To automate the monitoring of cooperative communication activities, one important issue is to find the proper receiver to deliver the question. There are mainly four communication forms defined in Caribou, in which a participant may be involved. Based on these four forms, we have designed our network personnel matching methods to match the pair(s) of people to have a communication channel.

Direct Personnel: In this type of communication, the questioner knows who should be asked for. The matching mechanism will simply use the pointed stuff name or ID to directly deliver the questioner's requests to those who are expected to answer.

Direct Task-individual: In this communication form, the questioner does not know who should be asked for. Whereas she may know what tasks are related to the concerned information that she is inquiring. The matching mechanism is to use the related tasks to trace the potential individuals who may have the answer. In a general case, if there is one task, then all the persons who have participated in that task may be considered as potential receivers. When there is more than one task related to the concerned question, then the matching mechanism is to select those persons who have participated in most of the tasks. It divides people into several groups according to the number of tasks they have participated. Those who have participated in more tasks will be considered as the most likely possible receivers.

Unknown Receptor: In some cases, the questioner may not be able to know who should be asked for, and even does not know which tasks are related with her question. The personnel matching mechanism will transfer this type of questions to e-bulletin board, group leader, technical coordinator, etc.

Public Informing: When the questioner just wants to provide some useful information/announcement for public, personnel matching module will convey it into all personnel.

The e-development personnel matching mechanism will ensure an answer of each question, as illustrated in Figure 7.10.

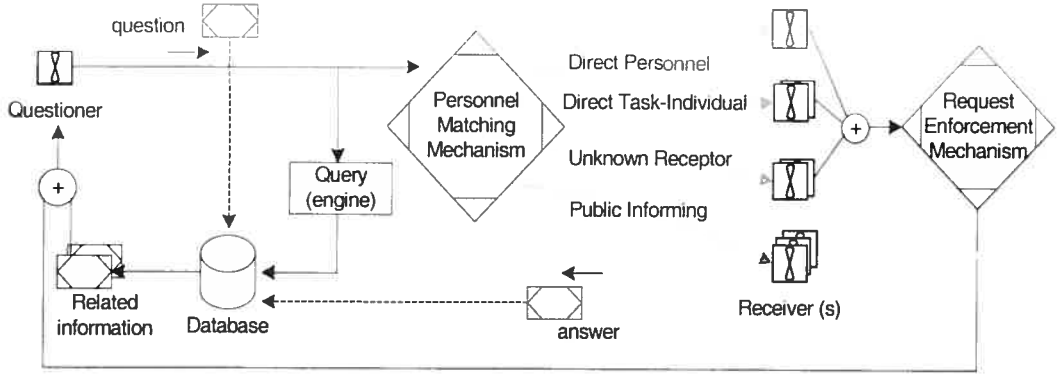


Figure 7.10: Caribou Personnel Matching Model: Question Transaction Process

When the questioner publishes a question in a collaborative e-development environment, firstly, the question itself will be recorded in a communication database; then, a search engine will query this database to find out related information concerning the question. Meanwhile, the personnel matching mechanism will build up a channel to the proper person(s). The request enforcement mechanism will ensure the elicitation of answers from those targeted receivers. The answer will also be recorded in the communication database for other reference usage. The feedback from the database and receivers will be presented to questioner.

The communication interaction may recur for several times, until the problem is solved or deadlocked. The automated control module will deal with those abnormal situations.

7.3.3 Request Enforcement: Performance Control

Now we will realize our second goal of securing the solicitation of desired information in e-development. That means, the answering of a question is no longer an option, or a spontaneous reaction.

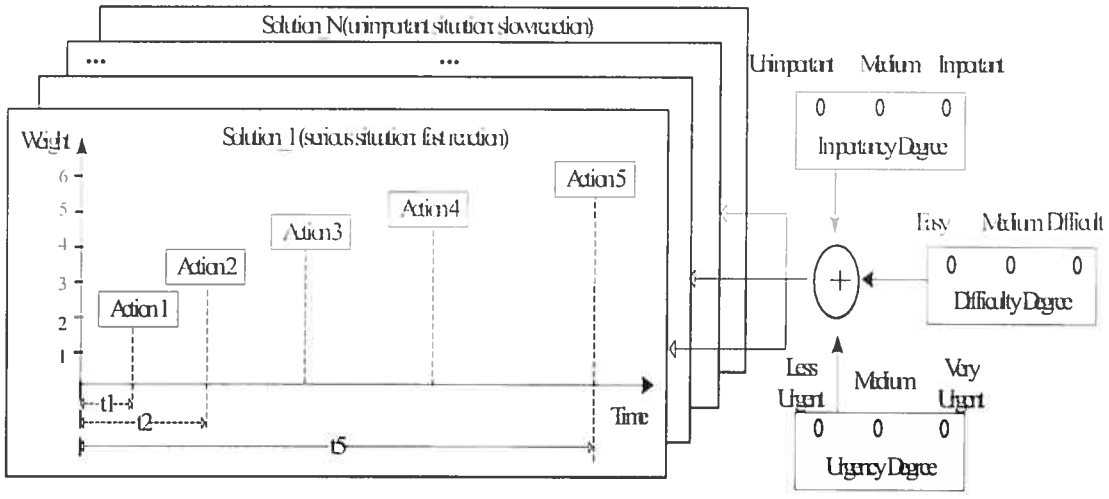


Figure 7.11: Request Enforcement Model

It is somehow a mandatory request in a typical software e-development project. To fulfill this objective, we have established a request enforcement mechanism to the help questioner squeeze out the satisfied answer, as showed in Figure 7.11. In our prototype, there is a set of pre-defined dealing solutions.

Each solution contains a collection of action scripts to invoke correspondent actions based on the pass-time length. It could be a remainder to the receiver after a short lap of time; and if the passed time gets longer, that action may be to inform the group leader, etc. For each question, before it has been sent out, the questioner has to mark estimated subjective values for three factors, namely urgency degree, importance degree, and difficulty degree. Then, based on these values, the *Caribou* communication control module will figure out the suitable solution, and trigger the corresponding actions based on the waiting time intervals. These actions are labeled with weight value. Each weight represents the tolerance degree attributed to the action when waiting interval exceeds a certain time-length of threshold.

The solution set (left) includes many solutions to deal with different types of conditions. For each solution, it includes several actions to cope with that condition based on waiting time intervals. The control module uses these three factors (right) to calculate the number of most suitable solutions.

Evaluating Communication Quality

The communication performance data will be recorded to quantify the evaluation of the communication quality, which is our third goal. A standard evaluation check form helps the receiver to evaluate the question. The questioner also evaluates the reply by simply checking the quality tabular. The inquiry transaction may repeat itself for several times until the problem is solved or dead-blocked. All these performances are recorded by the communication monitoring module. The result will be processed by automated control module to trigger the corresponding control actions. A dead-locked question will be prompted to a higher-level group leader or technical coordinator; widely concerned questions will be presented to project managers, and request her to provide a general solution or suggestion; an extra delay for a question will generate a caution message to the group leader, etc.

7.4 Summary

Web-based collaborative migration projects involve many people working together without the barrier of time and space differences. However, the large scale of collaboration in a typical migration project lacks of sufficient support techniques to facilitate project planning, monitoring of distributed collaboration tasks, and communication. In this chapter, we have presented our approach and the prototype of *Caribou*, a support environment for collaborative migration projects, to tackle these important issues. In addition, we also provide solutions to automate the dynamic control of communication to fulfill the goal of enhancing the collaboration performances.

Chapter VIII. Experiment and Evaluation

In this chapter, we present the experiments that we have conducted to empirically evaluate our migration approach. We have selected three medium size legacy systems from open source organizations as the case study subjects. They all are implemented by using procedure programming methodology, and have evolved for many years. Our experiments are conducted to migrate these systems into the Object-Oriented paradigm, and preserve their original functionalities. The renovated systems are supposed to benefit from object technology in terms of having higher quality in maintainability due to many desirable features provided by object technology. We apply our migration approach and techniques to conduct these experiments. The results in turn evaluate our approach, and illustrate directions for future improvements.

8.1 Experiment Suites

8.1.1 Case Study Subjects

The case studies are performed on three medium size open source procedural systems (written in C). They are distributed under the GNU General Public License. They are selected from different domains:

1) **"*Interest*"** is a finance management system for personal investments. It is designed to analyze individual stock market investment performance. It can be obtained from web site http://sourceforge.net/projects/interest

2) ***"GALOPPS"*** is a flexible Genetic Algorithm API that is optimized for portability and parallelism system. It is based upon Goldberg's Simple Genetic Algorithm (SGA) architecture. It can be obtained from http://garage.cps.msu.edu/software/galopps/index.html

3) ***"Expat"*** is a XML parser library. It is a stream oriented parser that requires setting handlers to deal with the structure that the parser discovers in the document. It can be obtained from: http://sourceforge.net/projects/expat/ Project home web site: http://www.libexpat.org/

Table 8.1 describes the main characteristics of these systems.

Legacy System	Lines of Code	Number of Files	Number of Functions	Number of User Defined Data Types	Number of Global Variables
Interest	28,748	96	551	220	122
Galopps	24,746	59	371	82	374
Expat	26,926	40	148	404	593

Table 8.1: Source Code Characteristics of the Examined Systems

8.1.2 Experiment Carrier

The case studies were conducted by students studying computer science at University of Montreal. They were at either undergraduate or graduate level. The experiments were carried out as course projects in course IFT6251- "Special Topics in Software Engineering". All of these students had at least one year of programming experience and were familiar with procedural programming languages and object-oriented programming languages. In the "Interest" project, 26 students participated in the experiment during a period of one month. In the "Expat" and the "Galopps" projects, 3 and 5 students participated in the experiments, respectively, during a period of two months.

8.2 Legacy System Analysis

One of the challenges for reverse engineering is to design and implement a parser to analyze static source code features. Even for a simple programming language, the effort to develop a parser could be very high. As an alternative approach adopted in this thesis, we use the open source software code analysis environment *Source-Navigator* to analyze the static source code features. *Source-Navigator* is published under GNU General Public License. It provides a build-in parser to parse source code and generates static code information repository. It supports C/C++, Java, Tcl, Fortran, Pascal, Ada, and COBOL.

We use our reverse engineering tools Dynamic-Analyzer and Collaboration-Investigator to analyze dynamic information of source code interaction and collaboration. The prototype of a migration project supporting system is also developed to conduct our studies of facilitating project planning and team collaboration. Table 8.2 shows the programming features of these tools.

Components	Line of Code	Implementation Language	Database
Dynamic -Analyzer	37,235	C#	Oracle 9i
Collaboration- Investigator	13,984	C#	Oracle 9i
Caribou Migration Supporting Environment	9,443	C#	Oracle 9i

Table 8.2: Experiment Tools

8.2.1 Collaboration Pattern and Conceptual Role Analysis

In this section, we present the experiment that evaluates how our approach supports the understanding and recovery of collaborations and roles in legacy systems. For the purpose of simplicity, we only demonstrate the analysis of the finance management system "Interest". Similar analyses were also conducted on the other two legacy systems.

Analysis question and hypothesis

The "Interest" system provides a bunch of tools to help users to analyze their stock investment performance. To better understand how these tools are implemented, we put

forward several questions and hypotheses to start our study. We notice that the code is divided into three major directories. Source root "src/" contains two subdirectories, namely "src/base/" and "src/widgets/" respectively. Meanwhile, the source file names under each directory have different characteristics. For example, the files under "/base" subdirectory have names like "color.c", "error.c", "transaction.c", etc. We thus hypothesize that the modules under different directories deal with different functional issues.

Questions:

- What's the relationship between these source code modules (files)? What are the roles and general functions they represent?
- Which modules work together to realize different system functionalities?
- How much contribution do they have?
- How do they cooperate with each other?

Hypotheses:

- We suspect that the modules follow a certain pattern in cooperating with each other in order to implement the system functionalities.
- We also guess that each module represents a certain system constructional concept, which reflects a role when they collaborate with each other.
- A module may have more than one role when it participates in different patterns, but it could have one major role in the whole system.

Recovering code collaboration patterns and roles

Rather than trying to understand the whole legacy software in one step, we study system behavior and its implementation based on individual system functionalities. In this case, we select the most interesting part of our target legacy system, the functionality of graphical analysis of stock investment. We would like to know if modules follow some patterns in their cooperation, and find out how they collaborate with each other. Furthermore, we also want to know if these patterns recur in other system functionalities

as well. Moreover, we would like to investigate which modules participate in the patterns, how they interact, what is the relationship among them, and what is the role of each module.

Recording dynamic information. We execute the target system with instrumented code. The selected scenario is focused on the system functionality of stock performance analysis. We use our reverse engineering tool *Dynamic-Analyzer* to collect the interaction information into repository, and visualizes the dynamic effects of module interactions. The scenario creates 68,123 function invocation events.

Setting pattern discovery criteria. We define the following four criteria: (1) select four out of six interaction components to observe, namely sender module, receiver module, invocated routine, and direction; (2) set the deepest invocation level at 18; (3) do not ignore self-interaction; (4) omit considerations of the invocation level when comparing two collaboration patterns.

Automatic pattern detection with *Dynamic-Analyzer*. We use the *Dynamic-Analyzer* to generate the initial fine-grained result of recovered patterns from the whole interaction serials. As shown in Figure 8.1, the total interaction sequence lasts for 85 visual screen frames.

Early results indicate only the three most important patterns contribute almost the whole of this system functionality. Based on the visual screen frame number, we can see that one pattern lies from 2 to 20 plus from 34 to 51 frames; another one lies from 21 to 31 plus from 52 to 61 plus from 63 to 74 frames, and the last one lies from 75 to 83 frames respectively. These three major collaboration patterns amount to over 90% of the interaction frames. More significantly, the participants are limited to less than 11 modules, compared with totally 94 modules of the whole system. This result shows that, although the "Stock analysis" subsystem has very complex system functionalities and various dynamic behaviors, with the help of collaboration pattern analysis, maintainers can quickly focus on studying several important patterns to understand the whole implementation.

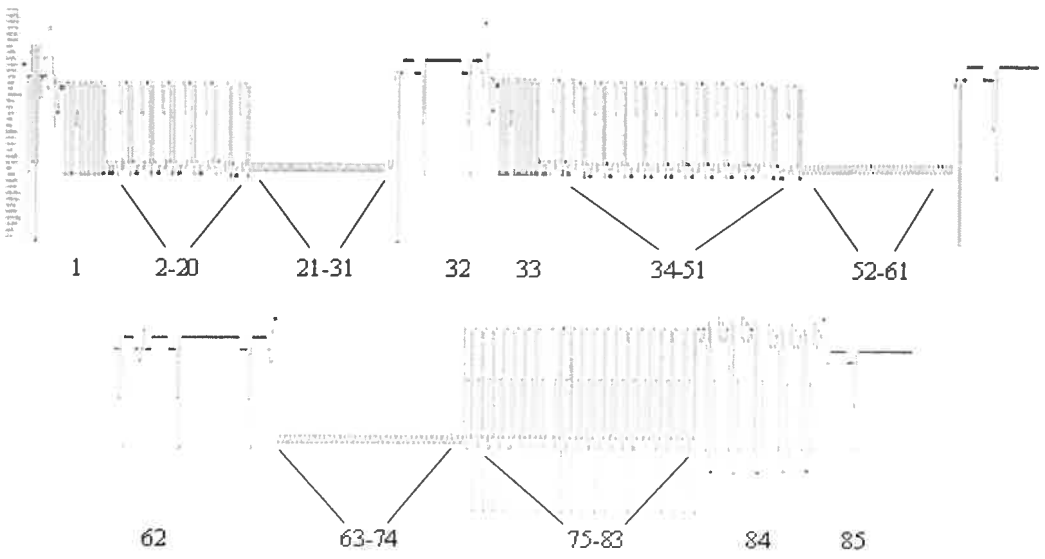


Figure 8.1: Automatic Collaboration Pattern Recovery (The number indicates frame id)

Deep study with *Collaboration-Investigator*. As the next step, we further analyze the details of collaboration patterns and the relations (roles) among the participants in each single pattern. We use our analysis tool *Collaboration-Investigator* to recover all the patterns that satisfy discovery criteria which have been settled formerly. A total of 9 patterns have been recovered. They are listed in Table 8.3. The name of each collaboration pattern is composed of its identity number, the first module's name, and the first routine name that invoked by that module. We notice that some patterns consist of several smaller patterns.

Recovered collaboration patterns
#01_account.c : ac_init_attributes
#02_import.c : parse_header_data
#03_transaction.c : trans_compare_by_date
#04_nodemenu.c : nodemenu_new
#05_menubar.c : analyze_flags
#06_configuration.c : clear_autoload
#07_intranslist.c : in_trans_list_set_account
#08_dataset.c : ds_show
#09_intransbut.c : in_trans_but_init

Table 8.3: List of Patterns

To answer the questions and verify the hypotheses, we now select one pattern to study in detail (see Figure 8.2). The No.7 pattern has one of the most significant repetitive characteristics among all the recovered patterns. It has 31 out of 85 visual screen frames with a nested pattern No.3. The following interaction fraction shows the retrieved components and collaboration instances from the *Collaboration-Investigator* tool. From the composition relationships illustrated in the segment, we can find that when module Intranslist.c sets transaction accounts, it in fact fills in the transaction list. To accomplish this job, it first asks module Translist.c to reset transaction list. Then it delegates the whole task to module Transaction.c. The latter one deals with the details of comparing and compiling work with the support from module Transarray.c.

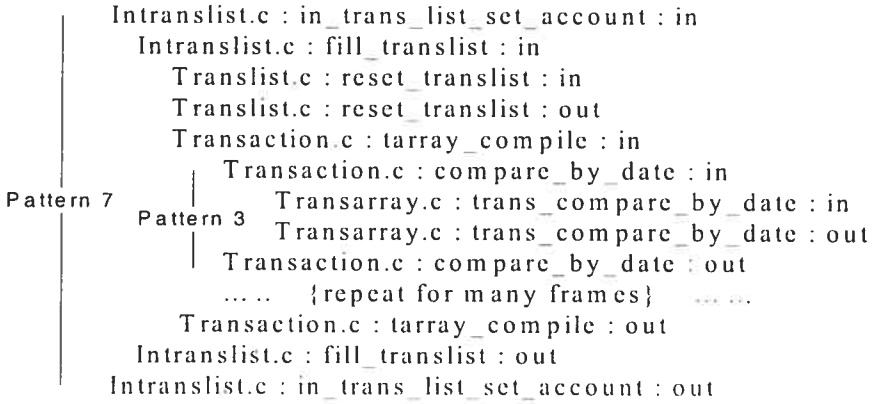


Figure 8.2: Detected Collaboration Pattern

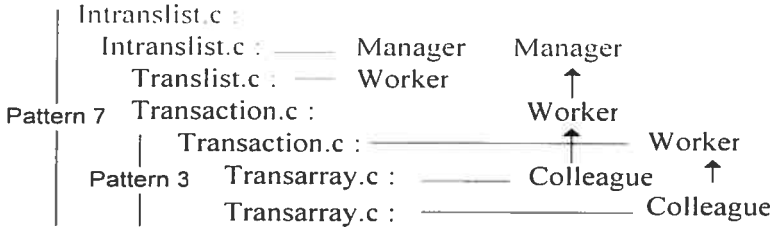


Figure 8.3. Module Conceptual Role Recovery

We further want to know the role of each module inside of pattern 7 and pattern 3. This time, we use the Collaboration-Investigator tool to inspect the role of each module. As described in Chapter 4, there are four pairs of role stereotypes defined in the tool. Based on the transaction information, each module in the pattern will be assigned at least one role stereotype.

In this case, there are three groups of roles assigned to modules, as shown in Figure 8.3. In collaboration pattern 3, modules Transaction.c and Transarray.c work as "Worker vs Colleague". That means, both of them collaborate fairly and equally to contribute the implementation of the system functionality. While module Intranslist.c works as a job distributor. It assigns tasks to Translist.c and Transaction.c, and the latter one cooperates with Transarray.c to fulfill most of the work. Further investigation shows that module Intranslist.c belongs to the root directory "/src/", whereas both modules "Transaction.c" and "Transarray.c" belong to subdirectory "/src/base/".

To better understand a single module's function in the scenario, the maintainer can choose that module, and query all the roles it plays inside of any collaboration pattern, thus to get a broad understanding of what kind of role that module has in general. In our case, module Transaction.c works as "Worker". This gives maintainers a strong suggestion that this module implements a certain system constructional concept, which means "labor". This module accomplishes the real job, and contributes to those who dispatch tasks to it.

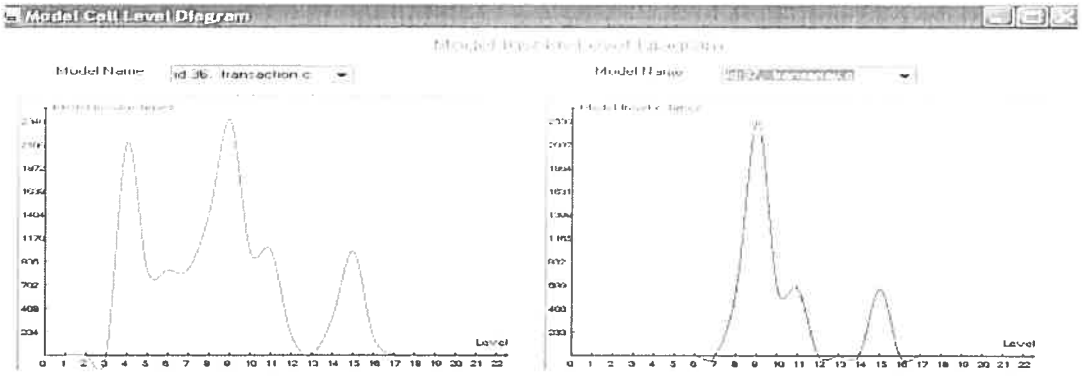


Figure 8.4: Analyzing two modules within a collaboration: Transaction (left) and Transarray (right)

Within a single collaboration pattern, we also use the *Dynamic-Analyzer* to analyze two modules which have strong cooperations, especially when these two modules have a certain type of role relationships within pattern.

Two modules are compared with the density of their interaction instances, the depth level, the activity frequency, and time period. Figure 8.4 shows the comparison of modules Transaction.c and Transarray.c. The vertical axis represents the number of invocation times; the horizontal axis represents the invocation depths. We can find that starting from depth 7, they have the same fluctuation pattern of invocation frequency and calling instances. This is confirmed by our previous collaboration pattern analysis, since these two modules cooperate as "Worker vs Colleague", and form collaboration pattern 3. The pattern occupies a relatively large segment in the whole interaction serials (31 out of 85 observing screen frames). This fact suggests that both modules have tight coupling with each other after depth level 7. But from depths 1 to 7, module Transarray.c has no interaction instances, while module Transaction.c is still active from depths 4 to 6. This indicates that, the module Transaction.c not only cooperates with Transarray.c, but also participates in other activities, which is later proved by further using our tools to trace it; while from depth level 7, the rest part of module Transaction.c only faithfully cooperates with module Transarray.c.

8.2.2 System Decomposition

System decomposition is important to limit the complexity and risk associated with the re-engineering activities of large legacy systems. In Chapter 4, we have discussed the techniques that we used to decompose a legacy system into smaller but cohesive parts. We use these techniques to illustrate the decomposition results of the "Expat" legacy system. Similar experiments are also conducted with systems "Interest" and "Galopps".

The "Expat" system is divided into three major parts, namely XML Token, XML Role, and XML Parser. Each part contains several modules which strongly cooperate with each other to form a subsystem. The decomposition result is illustrated in Table 8.4.

Subsystem 1 (XML Token) :

Encoding	Position
Utf8Encoding;	Token
Utf16LittleEndianEncoding	XmlTokException
Utf16BigEndianEncoding	XmlTokInvalidException
AsciiEncoding	XmlTokNoneException
UnknownEncoding	XmlTokPartialException
Latin1Encoding	XmlTokPartialCharException
EncodingInfo	XmlTokTrailingCrException
TextDeclaration	XmlTokTrailingRsqbException
XmlDeclaration	XmlTokOtherException

Subsystem 2 (XML Role) :

Entity	NS_ATT
ElementType	Binding
DTD	Tag
Prefix	Tag_Name
Attribute	PrologHandler

Subsystem 3 (XML Parser):

XML_Content	XMLStartElementEvent
XML_Features	XMLEndElementEvent
XML_Error	XMLCharacterDataEvent
XML_Version	XMLProcessingInstructionEvent
XML_Parser	XMLEndPrologEvent
XML_ParserStatus	XMLCommentEvent
ParserApplication	XMLStartCdataSectionEvent
Block	XMLEndCdataSectionEvent
HashTable	XMLStartEntityReferenceEvent
HashTableIter	XMLEndEntityReferenceEvent
Named	StringPool
OpenEntity	Content_Scaffold

Table 8.4: Decomposition of "Expat" Legacy System

Here we evaluate the quality of the techniques that we used in decomposing the legacy system. The evaluation consists of comparing the decomposition parts identified by each

technique to the final results obtained by programmer. For objectivity purpose, we use a comparison framework introduced by Girard et al. [Gir99].

Let C be the set of the identified decomposition parts and R be the set of reference parts. A part $c \in C$ is said to *approximately match* a part $r \in R$ ($c <<_p r$) if the modules of c matches the modules of r with a tolerance parameter p. For our evaluation, we set p to 0.7, which means that at least 70% of the modules of c are in r. This value of p was used by the original authors of the framework. Using the << relation, the framework proposes three possible categories for an identified system decomposition part c:

- **Good**: if $\exists\, r \mid c <<_{0.7} r$ ***and*** $r <<_{0.7} c$
- **Ok**: if $\exists\, r \mid c <<_{0.7} r$ **or** $r <<_{0.7} c$
- **False positive**: otherwise

Figure 8.5 visualizes the results of the comparison between the identified and final decomposition parts for each system.

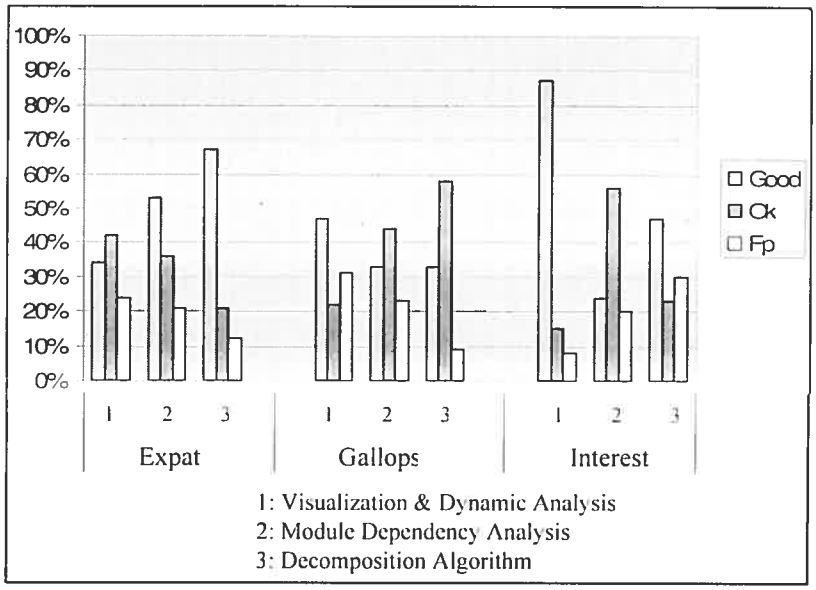


Figure 8.5: Summary of Evaluation Results

The performance of each technique varies according to the characteristics of the legacy system. The Visualization and Dynamic Analysis technique performs very well with the

"Interest" system. The reason is that the visualization of system inner code activities only reviews the structure and collaboration of code artifacts of an independent system. It is difficult to discover the structure of libraries. While both "Expat" and "Galopps" are library/API, they do not provide an application system for us to apply dynamic analysis techniques to observe its code activities. We can only use the applications that use their library/API to conduct the experiments. On the contrary, Module Dependency Analysis and Decomposition Algorithm perform equally very well in all these three systems. The programmers mainly use the suggestions provided by these three techniques, and construct the final decomposition of the legacy systems.

8.3 Object-Oriented Re-Architecturing

In Chapter 5, we have discussed the techniques that we used to elicit object models from legacy system. We use the system decomposition result as input, and apply these object identification techniques to generate candidate classes. The object models for each decomposition are established based on the candidate classes (See Figure 8.6).

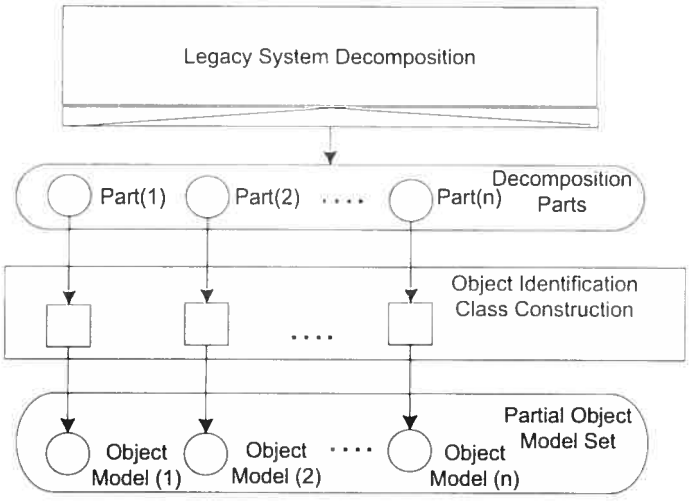
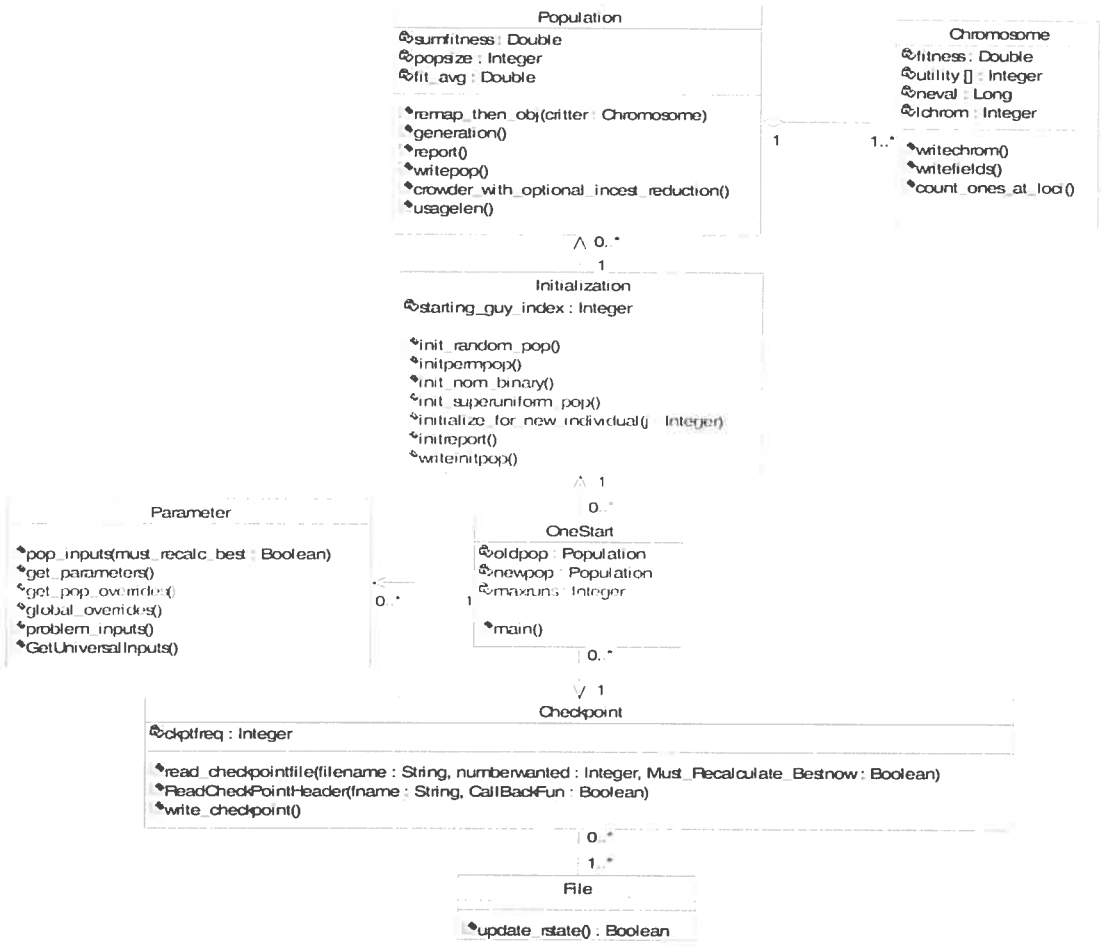


Figure 8.6: Object Model Elicitation

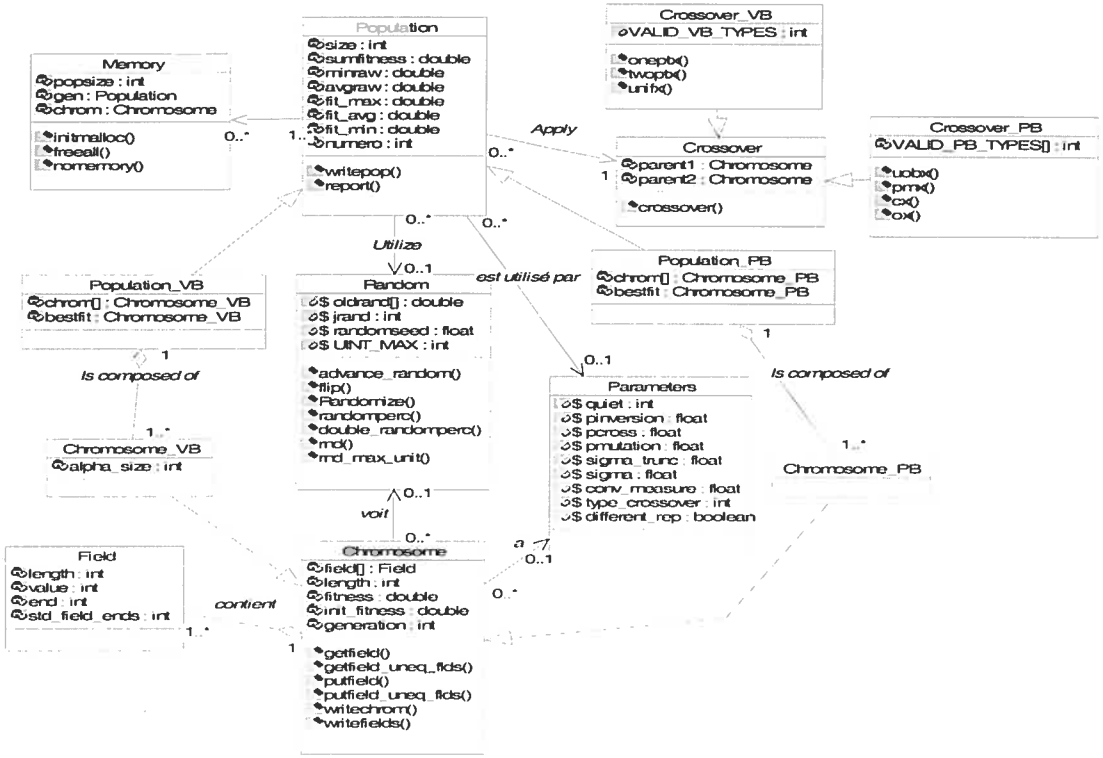
The input of object model elicitation is the outcome of the system decomposition. Different techniques discussed in Chapter 5 will be used to facilitate the object-oriented

modeling of the legacy system. We use "Galopps" as examine sample to illustrate the object modeling result of target system. Similar experiments are also conducted with "Interest" and "Expat" as well. We mainly applied "Rule Based Class Recovery" and "Dynamic Featuring" techniques on these three legacy systems to elicit object models. In the end, we evaluate the performance of those object identification techniques presented in Chapter 5.

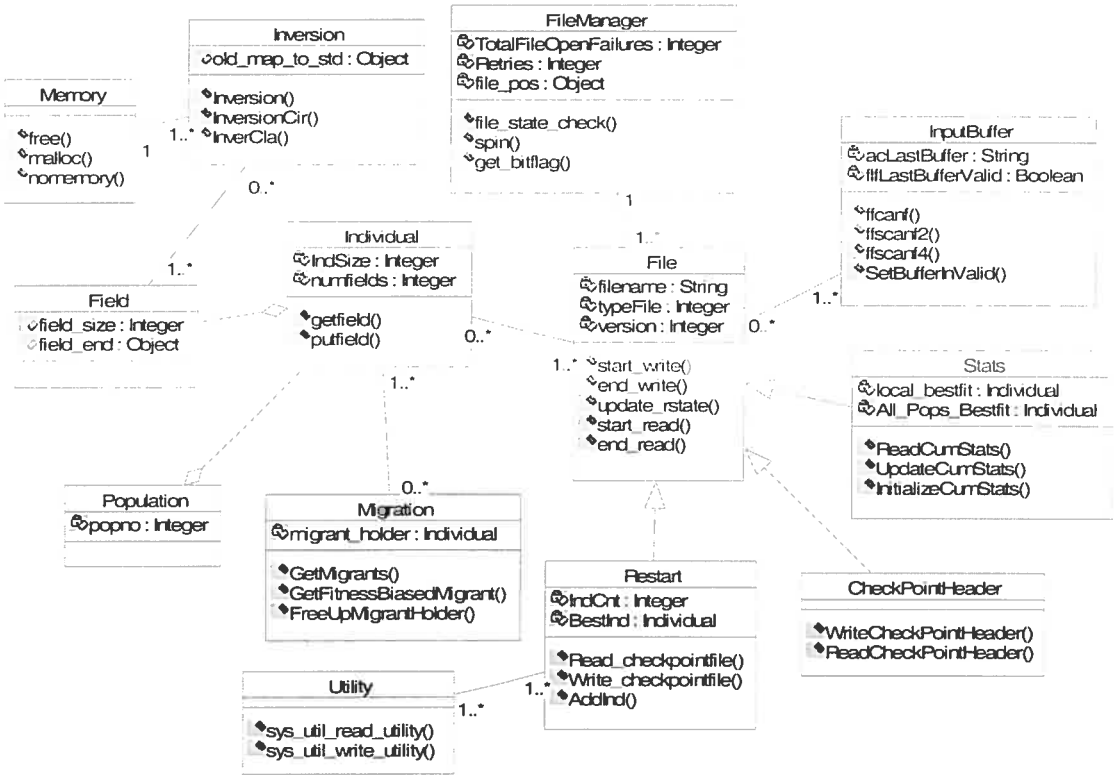
"Galopps" is a genetic algorithm API package. After the decomposition process, it is divided into four major parts: Initialization, Parameter Control, Application Construction, and Utility Tool. Each application that uses "Galopps" genetic algorithm API will apply various modules in each of these four parts to construct an integrated application system. Figure 8.7 illustrates the elicited object models of each decomposition part.



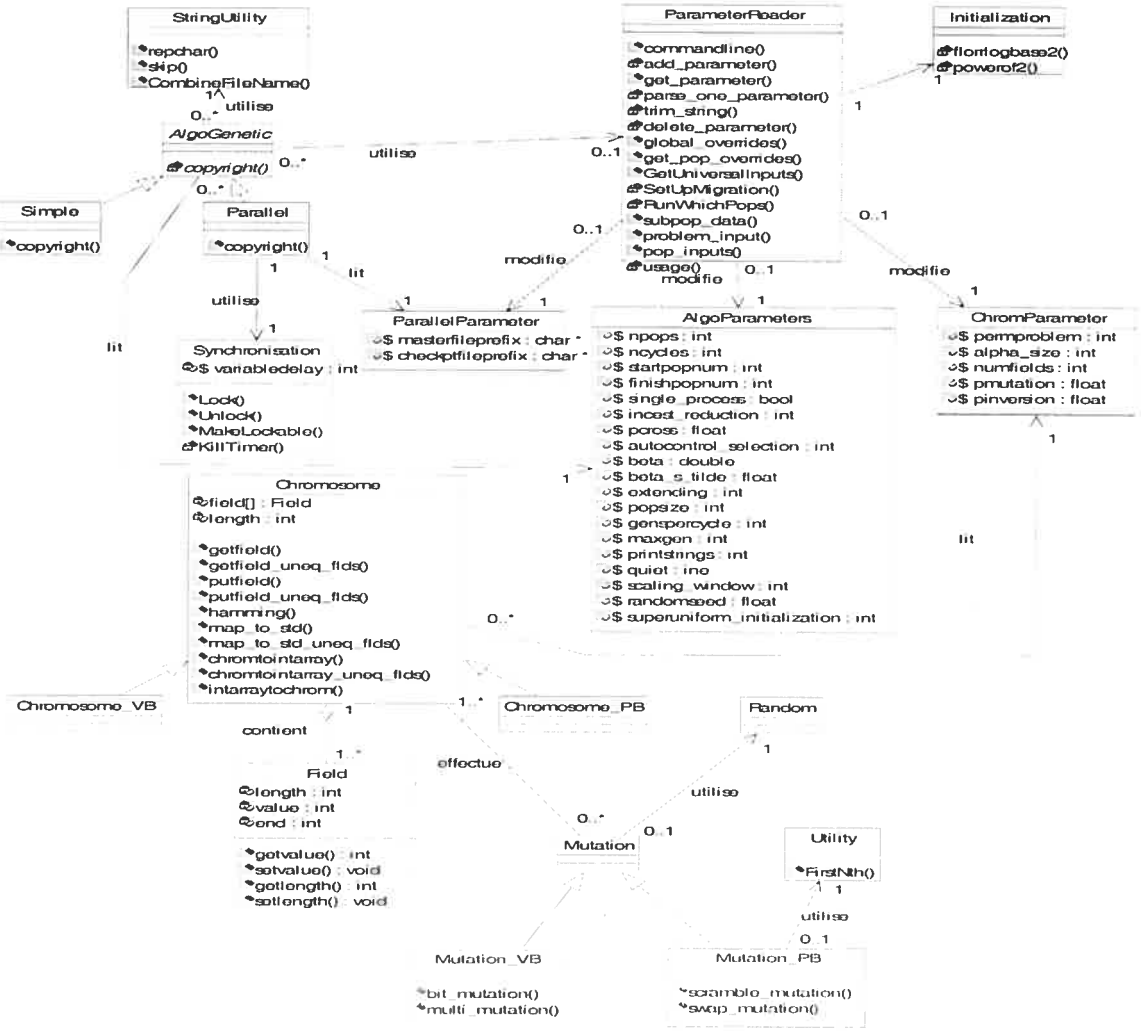
(i) Initialization Part



(ii) Utility Tools Part



(iii) Application Construction Part



(iv) Parameter Control Part

Figure 8.7: Object Modeling of "*Galopps*"

We mainly use three types of object identification techniques in eliciting object models, namely rule-based class recovery, static featuring technique, and dynamic featuring technique, respectively. To evaluate the performance of those object identification techniques in our experiments, we apply the same comparison framework discussed in section 8.2.2.

8.4 Migration Supporting System and Incremental Implementation

Aiming at providing a comprehensive e-development supporting environment, we have developed a prototype called *Caribou,* to systematically facilitate web-base migration project and coordinate team collaboration. *Caribou* also works as an integrated platform to accommodate various tools and techniques that we have applied in the migration project. The ultimate objective is to maximize the efficiency and the chance of success for any legacy system migration project. The *Caribou* supporting system works as a general platform to serve collaboration among participants. Meanwhile, individual participants use it as a way to either access project resources or collaborate with others without considering the time and space variance.

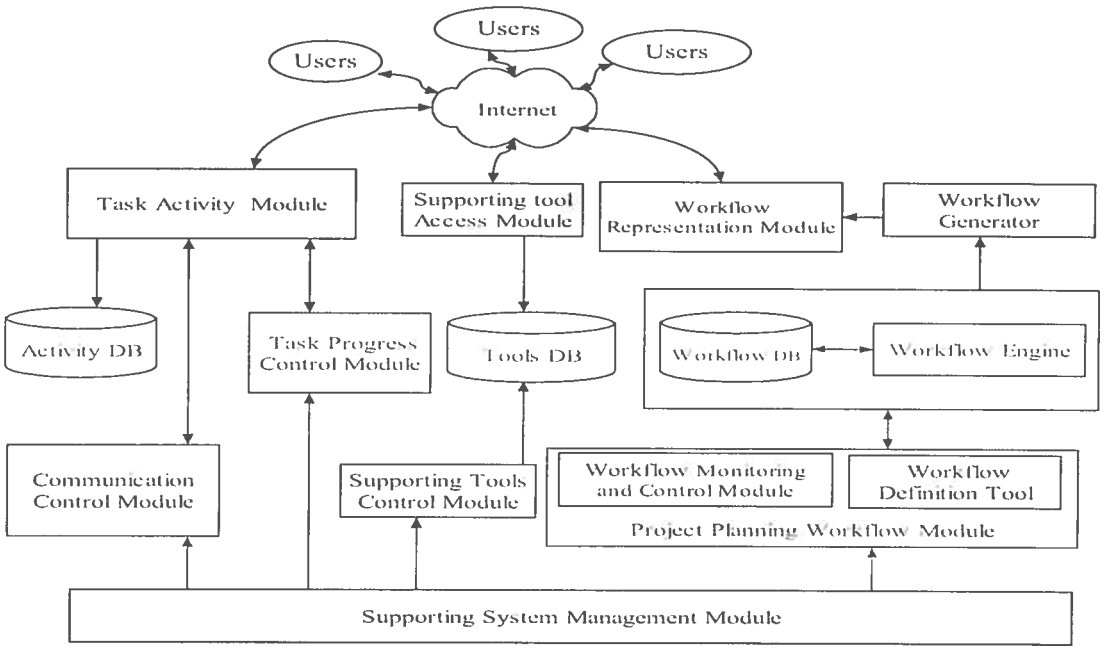
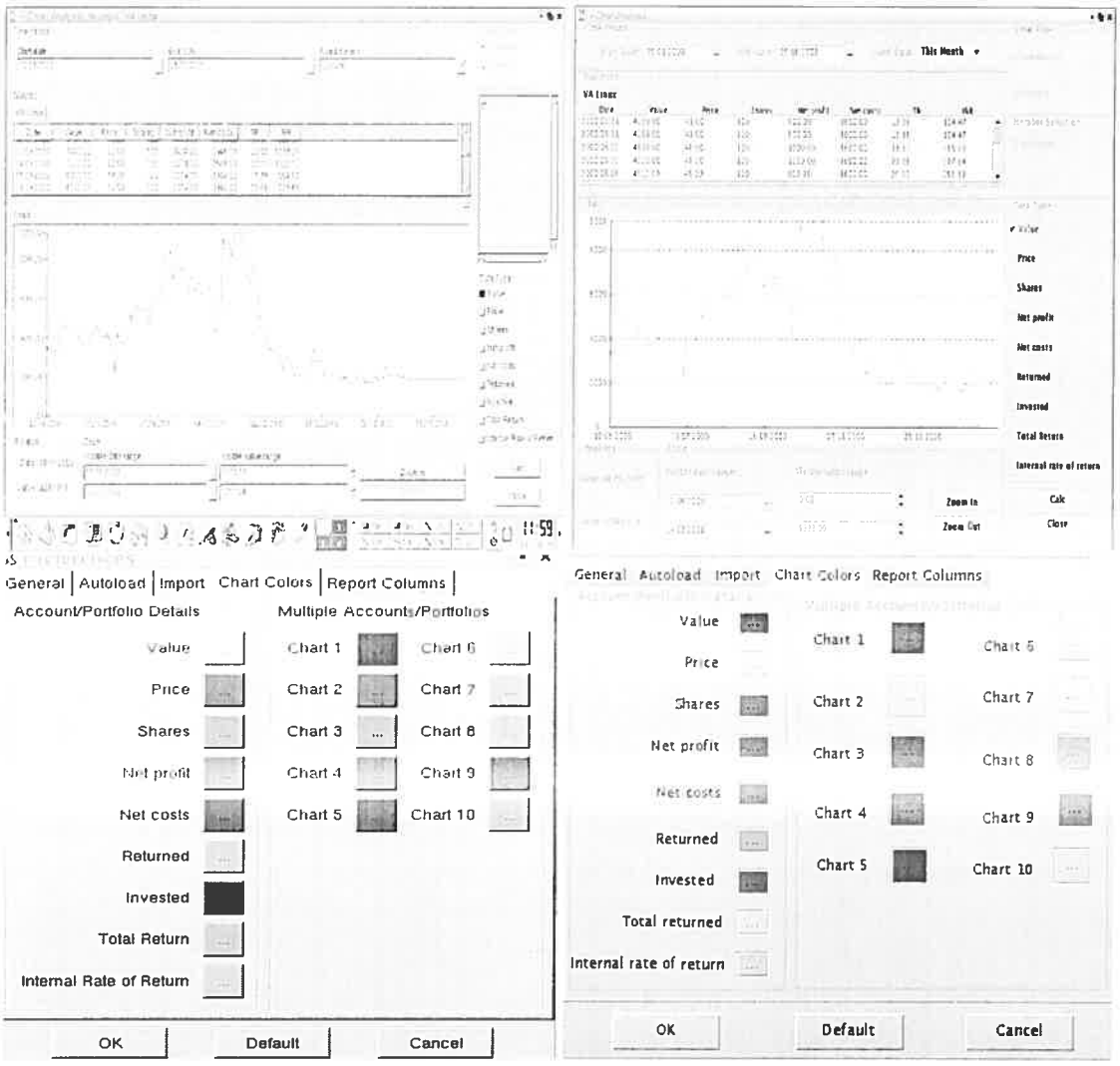


Figure 8.8: Caribou Environment Architecture

Figure 8.8 illustrates the architecture of Caribou. It includes five major parts, namely Supporting System Management Module, Communication Control Module, Task Progress Control Module, Supporting Tools Control Module, and Project Planning

Workflow Module. In the communication control module, a quantitative communication quality measurement mechanism is established to accommodate the need of e-development communication quality control. In this module, the performance of communication between developers will be enhanced through the communication control model and reinforcement model discussed in section 7.3. The major functionality of the task progress control module is to monitor the task progress in a web-based migration project. Supporting tools control module hosts the related tools that we have developed, or other third-party tools that will be used in a migration project.



(i) Legacy "Interest" System (ii)Migrated System Written in Java

Figure 8.9: The Legacy "Interest" System and Migrated OO System

Figure 8.9 shows the GUI comparison of the legacy "Interest" system, and the migrated OO system of it.

OO System	Lines of Code	Number of Files	Number of Member Functions	Number of Classes	Number of Global Variables
Interest	11,422	34	599	90	0
Galopps	8,572	42	312	42	0
Expat	8,371	66	397	32	0

Table 8.5: Features of Generated OO Systems

The incremental migration model is used to progressively implement partial of the target legacy system into OO paradigm. Table 8.5 shows the features of generated Object Oriented Systems, and…

8.5 Summary

In this chapter, we have presented the experiments that we have conducted to migrate legacy systems into the object-oriented paradigm. The legacy systems in our case studies are selected from open source organizations. They are all implemented by using procedural programming methodology, and have evolved for many years. We apply our migration approach and techniques to conduct these experiments. The experiments show promising results of successfully migrating legacy systems into an object-oriented platform with limited time period and human resources. The results in turn validate our approach and also indicate the potential improvement of our techniques. Due to the limitation of our access to large scale, real-life industry level legacy systems, our experiments are constrained to the available program resource provided by the open source community.

Chapter IX. Conclusion

In the field of software maintenance, legacy software systems are continuously evolved to meet the constant changing requirements. The demand of change may come from various resources, such as correcting errors, providing new functions, or porting to new platforms. To increase the maintainability of legacy systems and make them benefit the advantages of object-oriented technology, our goal in this research aims at providing a systematic migration methodology to help engineers to migrate legacy systems towards object-oriented technology.

This thesis presents our migration approach and techniques that we have developed to fulfill this goal. We have defined an ideal migration process model and practical migration process model to conduct the re-engineering practice. By applying dynamic analysis and software visualization, a reverse engineering technique is provided to analyze legacy systems. Several system decomposition techniques are developed to decompose a system into logically cohesive parts. We have also provided different techniques to recover object models from a legacy system. The migration process is divided into phases to carry out the progressive migration into practice. A prototype of a migration project supporting system is also developed to facilitate the collaborative migration activities.

9.1 Thesis Contributions

The proposed techniques and methodologies mainly facilitate the migration of legacy procedural systems into the object-oriented paradigm. The contributions of this thesis can be summarized as follows:

Development of a migration methodology framework ([Wu03b], [Wu05a])

Our research provides a comprehensive legacy migration methodology framework, in which various aspects of migration work have been addressed, and correspondent solutions have been provided.

Development of ideal and practical migration process models ([Wu05a])

We have developed both ideal and practical migration process models to accommodate the analysis and practice of a migration project. The ideal model is used to facilitate the understanding of detailed migration procedures that might be introduced in any legacy migration project, in which the migration condition is supposed to be ideal. The practical migration process model is designed to facilitate the normal migration practices which usually have many restrictions of resources.

Design and implementation of program analysis techniques (*Dynamic-Analyzer, Collaboration-Investigator*) ([Wu04a], [Wu04b], [Wu04c])

We have developed an approach to support the reverse engineering of legacy systems. It includes two major parts: the dynamic analysis of the legacy program and the visualization of software. The primary analysis mechanism is built upon two analysis subjects: the source code collaboration pattern and the conceptual roles of code artifacts.

Design and implementation of legacy system decomposition techniques ([Wu05a])

To facilitate our divide-and-conquer migration approach, we have designed three major legacy system decomposition techniques. They are adopted during the legacy system analysis stage to generate the migration units. The decomposition techniques smooth the course of incremental migration process.

Development of object modeling techniques for legacy systems ([Wu03a])

To conduct object-oriented re-architecturing work of legacy systems, we have designed several techniques to carry out the object identification job to construct the object model of target legacy systems.

Design of incremental migration model ([Wu03b])

To reduce the risk and complexity of large migration projects, we have developed an incremental migration approach to progressively migrate legacy system into object-oriented paradigm.

Development of migration project supporting system (*Caribou*) ([Wu05b], [Wu04b])

To support our research work, we have developed a re-engineering environment to facilitate the collaborative migration projects. It includes the support of migration project planning/scheduling, project task progress monitoring, collaborative communication and quality ensuring.

9.2 Future Work

The future work of this thesis can be continued in the following directions:

1. Generic Legacy System Migration Methodology Framework

   Currently, the migration methodology framework presented in this thesis is only discussing about the migration of procedural legacy systems into object-oriented technology. In the future, when more and more new technologies have emerged, and eventually when object-oriented technology becomes another kind of "legacy", we will face new questions of how to migrate them towards other more advanced technology? The generic migration methodology framework will address this concern in detail. We will further study more pertinent issues about legacy system evolution, and consistently benefit from more advanced computing technology.

2. Quality Enhancement in Legacy System Re-engineering

   Quality characteristics are very important to software systems. In fact, one of the major reasons for legacy system re-engineering practice is the consideration of quality improvement. How to systematically increase the desirable quality factors of subject systems should be a major concern of re-engineering projects. However,

in general, the migration approach only takes the advantages of acquiring attached quality improvements provided by migrated new technology platform. Therefore, how to construct an approach that specifically target desired quality characters during the migration process will become an important research issue.

3. Efficient Program Comprehension Model

   In this thesis, we have developed several program analysis techniques to facilitate the understanding of legacy systems. However, in our research, we found that there is a big distance between the full comprehension of a specific system and the techniques that are available both in academia and in industry. A future research direction should concentrate on the development of an efficient program comprehension model, which will both benefit the reverse engineering and re-engineering practice.

Bibliography

[Aal03] W.M.P. van der Aalst and A.H.M. ter Hofstede. "YAWL: Yet Another Workflow Language". *QUT Technical report, FIT-TR-2003-04,* Queensland University of Technology, Brisbane, 2003.

[Aeb97] D. Aebi, "Re-Engineering, Migration and Multi-Use of business Data", *Proceedings of the first International Conference on Software Maintenance and Reengineering,* IEEE Computer Society press 1997.

[Amb00] Scott W. Ambler, "Lessons in Agility from Internet-Based Development", IEEE Software, March 2002.

[And00] A.T Andrey., and C. Verhoef, "The Realities of Language Conversions", IEEE Software, November, 2000.

[Bat98] M. Battaglia, G. Savoia, and J. Favaro., "RENAISSANCE: A Method to Migrate from Legacy to Immortal Software Systems", *Proceedings of CSMR'98,* IEEE Computer Society, 1998, pp.197-200.

[Bal01] F. Balmas and Harald Wertz, "Identifying Information Needs for Program Understanding: an Iterative Approach", *Proceedings of 7th International Conference on Reverse Engineering for Information Systems,* Lyon (France) July 2001.

[Bal99] T. Ball, "The Concept of Dynamic Analysis". *Proceedings of ESEC/FSE,* LNCS, 1999, pp. 216-234

[Bas03] L. Bass, P. Clements, R. Kazman, "Software Architecture in Practice" (2nd edition), Addison-Wesley 2003.

[Ber00] J. Bergey, D. Smith, N. Weiderman, and S. Wood. "Options analysis for reengineering (OAR): Issues and conceptual approach". *Technical Note, Carnegie Mellon Software Engineering Institute* CMU/SEI, TN, 2000.

[Ben00] K. H. Bennett, V.T Rajlich, "Software Maintenance and Evolution: a Roadmap", "The Future of Software Engineering", Anthony Finkelstein (Ed.), ACM Press 2000.

[Ben95] K.H.Bennett, "Legacy Systems: Coping With Success", *IEEE Software,* January 1995, Vol 12, No. 1, pp 19-23.

[Bey01] Beyer, D.; Lewerentz, C.; Simon, F. "Impact of Inheritance on Metrics for Size, Coupling, and Cohesion in Object-Oriented Systems". Dumke/Abran: New Approaches in Software Measurement, LNCS 2001, Springer Publ., 2001, pp. 1-17

[Bia00]. A.Bianchi, D.Caivano, G.Visaggio, "Method and Process for Iterative Reengineering Data in Legacy System", *Proc. IEEE Working Conference on Reverse Engineering,* November 2000.

[Big94] T. Biggerstaff, B. Mitbander, and D. Webster. "Program understanding and the concept assignment problem". Communications of the ACM, 37(5):72–83, May 1994.

[Bis94] G. Biswas, J.B. Weinberg, C. Li, "ITERATE: A Conceptual Clustering Method for Knowledge Discovery in Databases". *Innovative Applications of Artificial Intelligence in the Oil and Gas Industry.* B. Braunschweig and R. Day, Editor. Editions Technip, 1994.

[Bri02] Liam O'Brien Christoph Stoermer, Chris Verhoef, "Software Architecture Reconstruction: Practice Needs and Current Approaches", *Carnegie Mellon Software Engineering Institute, Technical Report,* CMU/SEI-TR-024, 2002

[Bow98] R.W.Bowdidge,W.G.Griswold. "Supporting the restructuring of data abstractions through manipulation of program visualization". *ACM Transactions on Software Engineering and Methodology*, 7(2), April 1998.

[Bra98] van den Brand, M., Klint, P., and Verhoef, C. "Term Rewriting for Sale". *Electronic Notes in Theoretical Computer Science.* Elsevier Science B.V. 1998.

[Bro95] M.Brodie and M.Stonebraker. Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach. Morgan Kauffman, 1995.

[Byr92] E. J. Byrne. "A Conceptual Foundation for Software Re-Engineering". *International Conference of Software Engineering,* pp. 226-235, 1992

[Can00] G. Canfora, A. Cimitile, A. De Lucia, G.A. Di Lucca, "Decomposing Legacy Programs: A First Step Towards Migrating to Client-Server Platforms", *The Journal of Systems and Software,* vol. 54, 2000, pp. 99-110.

[Can01] G. Canfora, A. Cimitile, A. De Lucia, G.A. Di Lucca, "Decomposing Legacy Systems into Objects: An Eclectic Approach", *Information and Software Technology*, vol. 43, no. 6, 2001, pp. 401-412.

[Can99]G. Canfora, A. De Lucia, G.A. Di Lucca, "An Incremental Object-Oriented Migration Strategy for RPG Legacy Systems", *International Journal of Software Engineering and Knowledge Engineering, vol. 9, no. 1, 1999 pp. 5-25.*

[Car99] S. J. Carriere, S. G. Woods, R. Kazman, "Software Architecture Transformation", *Proceedings of WCRE 99, (Atlanta, GA),* October 1999, 13-23.

[Cha02] S. M.Charters, C. Knight, N. Thomas, M.Munro: "Visualisation for informed decision making; from code to components". *Proceedings of the 14th international conference on Software engineering and knowledge engineering,* July 15-19, 2002, Ischia, Italy. ACM, p765-772 2002

[Chi90] E.J. Chikofsky and J.H. Cross. "Reverse engineering and design recovery: A taxonomy". *IEEE Software*, 7(1):13-17, 1990.

[Chi94] S. Chidamber and C. Kemerer. "A metrics suite for object-oriented design." *IEEE Transactions Software Engineering*, 20(6):476–493, 1994.

[Cim95] A. Cimitile, G. Visaggio, "Software Salvaging and the Call Dominance Tree", *Journal of Systems and Software,* 28, pp. 117-127, 1995

[Cim98]A. Cimitile, U. De Carlini, A. De Lucia, "Incremental Migration Strategies: Data Flow Analysis for Wrapping", *Proceedings of 5th IEEE Working Conference on Reverse Engineering*, Honolulu, Hawaii, USA, 1998, IEEE Comp. Soc. Press, pp. 59-68.

[Cim99]A. Cimitile, A. De Lucia, G.A. Di Lucca, A.R. Fasolino, "Identifying Objects in Legacy Systems Using Design Metrics", The Journal of Systems and Software, vol. 44, no. 3, 1999, pp. 199-211.

[Cor90] T. A. Corbi , "Program Understanding: Challenge for the 1990s". *IBM Systems Journal,* 28(2):294-306, 1989

[Cor01] J. R. Cordy, T.R. Dean, A.J. Malton, and K.A.Schneider. "Software Engineering by Source Transformation — Experience with TXL". *In Proceedings of the 1st International IEEE Workshop on Source Code Analysis and Manipulation*, pages 168–178, Florence, Italy, November 2001.

[Coy00] F.P. Coyle "Legacy Integration Changing Perspectives", *IEEE Software*, Vol. 17 No. 2, March/April 2000, IEEE Computer Soc. Press, pp. 37-41.

[Dao04] M. Dao, M. Huchard, M. H. Rouane, C. Roume, and P. Valtchev, "Improving generalization level in uml models: Iterative cross generalization in practice," *in Proceedings of the 12th International Conference on Conceptual Structures (ICCS'04),* 14, Springer-Verlag, LNCS 3127, July 2004.

[Der03] J. Derome, K. Huang, "Creating and Delivering Value with Collaborative Software development", *Business Applications & Commerce* , September 2003

[Deu99] Arie Van Deursen, Tobias Kuipers. "Building Documentation Generators". *In Proceedings International Conference on Software Maintenance (ICSM 99).* IEEE Computer Society, pp. 40-49, 1999.

[Deu01] Arie van Deursen, "Recovering Rationale", *In Working Conference on Reverse Engineering WCRE'01,* Discussion Forum,. *IEEE Computer Society,* 2001

[Ede03]A. Eden, R. Kazman, "Architecture, Design, and Implementation", *Proceedings of the 25th International Conference on Software Engineering (ICSE 25)*, (Portland, OR), pp. 149-159, May 2003.

[Elo02] J. Eloff, "*Software Restructuring: Implementing a Code Abstraction Transformation*", *ACM International Conference Proceedings of SAICSIT* 2002,

[Fal98] E. Falkenauer. Genetic Algorithms and Grouping Problems. John Wiley and Sons, Chichester, 1998.

[Fan99] R. Fanta, V. Rajlich, "Restructuring legacy C code into C++", *Proc. IEEE Int. Conf. On Software Maintenace,* 1999.

[Fiu96] Fiutem, R., Tonella, P., Antoniol, G., and Merlo, E., "A Cliche-based Environment to Support Architectural Reverse Engineering", *Proc. of the International Conference on Software Maintenance.* pp. 319-328, 1996

[Fis87] D. H. Fisher, "Knowledge Acquisition via Incremental Conceptual Clustering", Machine Learning, Vol. 2, 1987, pp. 139-172.

[Fow99] Martin Fowler. Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[Gam95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software." Addison-Wesley, 1995.

[Gan99] B. Ganter and R.Wille. "Formal Concept Analysis: Mathematical Foundations". Springer, 1999.

[Gir99] J-F. Girard, R. Koschke, and G. Schied, "A Metric-based Approach to Detect Abstract Data Types and Abstract State Encapsulation", Journal on Automated Software Engineering, 6(4), 1999, pp. 357-386.

[Gnu00] GNU Project: Source-Navigator collaborative project, http://sourcenav.sourceforge.net/contrib.html, 2000

[God98] R. Godin, H. Mili, G. Mineau, R. Missaoui, A. Arfi, and T. Chau, "Design of Class Hierarchies Based on Concept (Galois) Lattices," Theory and Practice of Object Systems 4(2), 1998.

[Gol89] David E. Goldberg "Genetic Algorithms in Search, Optimization & Machine Learning", Addison Wesley, 1989.

[Gon98] N. A. Gonzalez, C. Czarnecki, and S. Pidaparthi, "Migrating Software from Procedural to Object-Oriented Architecture," *Proceedings of International Conference on Systems,Man and Cybernetics*, pp. 4872 –4877, San Diego, CA, October 1998.

[Gri91]Griswold, W. G. "Program restructuring as an aid to software maintenance". *Ph.D. thesis, Department of Computer Science and Engineering*, University of California, San Diego, 1991

[Gui86] J. Guigues and V. Duquenne, "Familles minimales d'implications informatives r´esultant d'un tableau de donn´ees binaires," *Math´ematiques et Sciences Humaines 95*, pp. 5–18, 1986.

[Hal88] HALL, R.P. "Seven Ways to Cut Software Maintenance Costs (Digest)", *in PARIKH, G.: Techniques of Program & System Maintenance (QED Information Sciences Inc., 1988 2nd edition*

[Hau01] M.L.Hause, M.R.Woodroffe, "Team Performance Factors in Distributed Collaborative Software Development", *13th Workshop of the Psychology of Programming Interest Group*, Bournemouth UK, April 2001.

[Har90] Mehdi T. Harandi, Jim Q. Ning, Knowledge-Based Program Analysis, *IEEE Software*, v.7 n.1, Pp.74-81, January 1990.

[How02] D. Howe (ed.), "The Free On-line Dictionary of Computing", http://wombat.doc.ic.ac.uk/ , 2002

[Huc02] M. Huchard, C. Roume, P. Valtchev, "When concepts point at other concepts: the case of uml diagram reconstruction," *in Proceedings of the 2nd Workshop on Advances in Formal Concept Analysis for Knowledge Discovery in Databases (FCAKDD)*, pp. 32–43, 2002.

[Jac00] Daniel Jackson and Martin Rinard. "Software Analysis: a Road Map", "The Future of Software Engineering", Anthony Finkelstein (Ed.), ACM Press 2000.

[Jer97] D.Jerding, S.Rugaber, "Using visualization for architectural localization and extraction", *proceedings WCRE, IEEE Computer Society, 1997, pp.56-65*

[Jes99a] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson."Legacy Information Systems: Issues and Directions" *IEEE Software,* September/October 1999.

[Jes99b]. Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. "Legacy Information System Migration: A Brief Review of Problems, Solutions and Research Issues".. *IEEE Software,* September/October 1999.

[Kaz99] Rick Kazman, S. Jeromy Carrière, "*Playing Detective: Reconstructing Software Architecture from Available Evidence"*, Journal of Automated Software Engineering, 6:2, April, 1999, 107-138.

[Kie02] Holger Kienle, Anke Weber, and Hausi Muller, "Leveraging SVG in the Rigi Reverse Engineering Tool", *SVG Open / Carto.net Developers Conference,* Zurich, Switzerland, July 2002.

[Kim00] H. S. Kim and J. Bieman, "Migrating legacy systems to CORBA based distributed environments through an automatic wrapper generation technique." *Proc. Joint meeting of the 4th World Multiconference on Systemics, Cybernetics and Informatics (SCI'2000) and the 6th International Conference on Information Systems Analysis and Synthesis (ISAS'2000),* June, 2000.

[Kos02] Rainer Koschke, "Atomic Architectural Component Recovery for Program Understanding and Evolution", *In Proceedings of the International Conference on Software Maintenance,* Montréal, Canada, October 2002.

[Kno03] Idrissa Konkobo, "Idenfificaiton des Objects dans les Applications Léguées basée sur les Algorithmes génétiques", *Master degree thesis*, University of Montreal, Computer Science Department, 2003

[Kui00] Tobias Kuipers and Leon Moonen. "Types and Concept Analysis for Legacy Systems". *In Proceedings of the International Workshop on Programming Comprehension (IWPC 2000).* IEEE Computer Society, June 2000.

[Lak97] A. Lakhotia. "A unified framework for expressing software subsystem classification techniques". *Journal of Systems and Software,* pages 211–231, March 1997.

[Lan03] Michele Lanza, Stephane Ducasse. "Polymetric Views - A Lightweight Visual Approach to Reverse Engineering". *IEEE Transactions on Software Engineering (TSE),* Vol. 29, No. 9, pp. 782 - 795, September 2003.

[Lau01] Y.T. Lau, *The Art of Objects: Object-Oriented Design and Architecture*, Addison-Wesley, 2001.

[Leh85] M.M.Lehman and L.A.Belady, "Program Evolution", *Academic Press, new York, 1985*

[Let99] A.N. Lethbridge, "Recovering software architecture from the names of source files", *Journal of Software Maintenance: Research and Practice,* November, 1999, pp. 201-221.

[Li00]. M. Li, O. F. Rana, M. S. Shields, D. W. Walker, "A Wrapper Generator for Wrapping High Performance Legacy Codes as Java/CORBA Components," *SuperComputing 2000, Dallas, USA, November 2000.*

[Lin97] C. Lindig, G. Snelting. "Assessing modular structure of legacy code based on mathematical concept analysis". *In 19th International Conference on Software Engineering,* ICSE-19, pages 349–359. ACM Press, 1997.

[Leo98]Cornelius T. Leondes, Fuzzy Logic and Expert Systems Applications, Academic Press, 1998

[Liu90] S. S. Liu, N. Wilde, "Identifying objects in a conventional procedural language: an example of data design recovery", *Proceedings of the Conference on Software Maintenance,* San Diego, California, November 1990, 266-271.

[Liv92] P.E. Livadas and P.K. Roy, Program dependence analysis, *International Conference on Software Maintenance*, 1992

[Liv94] P. E. Livadas, T. Johnson, "A new approach to finding objects in programs", *Journal of Software Maintenance: Research and Practice,* June, 1994, 249-260.

[Luc97] A. De Lucia, G.A. Di Lucca, A.R. Fasolino, P. Guerra, S. Petruzzelli, "Migrating legacy Systems towards Object-Oriented Platforms", *Proceedings of IEEE International Conference on software Maintenance, Bari,* Italy, Sept. 1997. IEEE Comp. Soc.Press, pp122-129

[Luc00]. A. De Lucia, G.A. Di Lucca, G. Canfora, A. Cimitile, "Decomposing Legacy Programs: A First Step Towards Migrating to Client-Server Platforms", *The Journal of Systems and Software, vol. 54, 2000, pp. 99-110.*

[Lud02] Martin Ludger, Anke Giesl, Johannes Martin. "Dynamic Component Program Visualisation". *In Proceedings of the 9th Working Conference for Reverse Engineering,* Richmond, Virginia, October 2002.

[Man98] S. Mancoridis, B.S. Mitchell, Y. Chen, E.R. Gansner. "Using automatic clustering to produce high-level system organizations of source code". *In the Proceedings of International Workshop on Program Comprehesion.* IEEE, 1998.

[Mal01] A.J. Malton. "The Software Migration Barbell". First ASERC Workshop on Software Architecture, August 2001.

[Mar01] I. Marsic, "An architecture for heterogeneous groupware applications". *Proc. 23rd IEEE/ACM International Conf. on Software Engineering (ICSE 2001)*, Toronto, Canada, May 2001.

[Mar02] J.Marin, H. A. Muller. "C to java migration experiences". *In the Proceedings of Sixth European Conference on Software Maintenance and Reengineering*, Hungary, 2002.

[Mel00]. Melody M. Moore, Lilia Moshkina, "Migrating Legacy System to the Internet" , *Working Conference on Reverse Engineering* 2000.

[Men04] Tom Mens, T.Tourwe, "A survey on software refactoring", *Transactions on Software Engineering*, IEEE Computer Society Press, February 2004.

[Mic03] Isabel Michiels, Dirk Deridder, Herman Tromp and Andy Zaidman, "Identifying Problems in Legacy Software", *Elisa ICSM workshop* 2003

[Mic80] R. S. Michalski, "Knowledge Acquisition through Conceptual Clustering: A Theoretical Framework and an algorithm for Partitioning Data into Conjunctive Concepts". *Policy Analysis and Information Systems*, 4(3), 1980, pp. 219-244.

[Mic83] R.S. Michalski, R. Stepp, "Learning from Observation: Conceptual Clusterings". In R.S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), Machine Learning: An artificial Intelligence Approach. Volume I, Morgan Kaufman, 1983.

[Mor98] Morgenthaler, J. D. 1998. "Building an Efficient Software Manipulation Tool". *Tech. Rep. Department of Computer Science, Hong Kong University of Science & Technology*, HKUST-CS98-02, January, 1998

[Mul00] H. Müller, J. Jahnke, D. Smith, M.-A. Storey, S. Tilley, and K. Wong. "Reverse Engineering: A Road Map", "The Future of Software Engineering", Anthony Finkelstein (Ed.), ACM Press 2000.

[May95] Anneliese von Mayrhauser and A. Mary Vans. "*Program Comprehension During Software Maintenance and Evolution.*" IEEE Computer. August, 1995. Vol. 28, No. 8, pages 44-55.

[Mow95] T. J. Mowbray, R. Zahavi. "The Essential CORBA: Systems Integration Using Distributed Objects". John Wiley Sons Inc., 1995.

[Nar00] B. Nardi, S.Whittaker, E.Bradner, (2000): "Interaction and Outeraction: Instant Messaging in Action", *Proceedings of CSCW 2000*, Philadelphia PA, December ACM Press, 2000, pp79-88

[New95] P. Newcomb, G. Kottik. "Reengineering procedural into object-oriented systems" *.In Second Working Conference on Reverse Engineering;* WCRE'95, pages 237–249. IEEE Computer Society, 1995.

[Nes02] S.V. Ness "Using NDoc: Adding World-Class Documentation to Your .NET Components", *ONDotNet.Com,* O'Reilly, Dec 2002

[Nie99] Flemming Nielson, Hanne Riis Nielson, Chris Hankin: *Principles of Program Analysis.* Springer, 1999

[Not02] David Notkin, "Longitudinal program analysis", ACM SIGSOFT Software Engineering Notes , *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering,* Volume 28 Issue 1, November 2002

[Oga94] R. M.Ogando,Yau, S. S., Liu, S. S., Wilde, N., "An object finder for program structure understanding in software maintenance", Journal of Software Maintenance: Research and Practice June 1994, 261-283.

[Omg01] OMG, "Unified Modeling Language (UML) specification 1.4", *http://www.omg.org/technology/documents/formal/uml.htm 2001*

[Pal97] S.Palthepu, J.Greer, G.McCalla, "Cliche recognition in legacy software: A scalable, knowledge-based approach". *IEEE Working Conference on Reverse Engineering*, IEEE Comp. Soc. Press, Oct 1997. pp. 94—103

[Pao01] T. Paolo, "Concept Analysis for Module Restructuring", *IEEE Transactions on Software Engineering,* vol. 27, n. 4, pp. 351-363, April 2001.

[Par98] W. J. Park, S. Y. Min and D. H. Bae, "Object-Oriented Model Refinement Technique in Software Reengineering," *Proceedings of the 22nd International Computer Software and Applications Conference*, pp. 32-38, Vienna, Austria, August 1998.

[Pat99]. P. Patil, et al, "Migration of Procedural Systems to Network-Centric Platforms", *CASCON 1999.*

[Pau94] S.Paul, A.Prakash, "A Framework for Source Code Search Using Program Patterns", *IEEE Transactions on Software Engineering* 20(6), 1994, pp. 463-475

[Pau98] W.D.Pauw, D.Lorenz, J.Vlissides, and M.Wgman, "Execution patterns in object-oriented visualization", *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS'98), USENIX, 1998, pp. 219-234*

[Pen02] J. Peng and K. H. Law, "A Prototype Software Framework for Internet-Enabled Collaborative Development of a Structural Analysis Program", *Engineering with Computers, Springer-Verlag,* 2002, 18: pp. 38–49

[Pre01] Roger S. Pressman "Software Engineering: A Practitioner's Approach", McGraw-Hill Science/Engineering 2001.

[Pre00] N. L. Preguiça, J.Martins, H., Domingos, S. Duarte, "Data management support for asynchronous groupware". *Proc. ACM Conference on Computer-Supported Cooperative Work (CSCW'00)*, Philadelphia, PA, December 2000, pp.69-78

[Pri93] B. A. Price, R. M. Baecker, and I. S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[Rai00] K. Rainer, "Atomic Architectural Component Recovery for Program Understanding and Evolution", *Ph.D Thesis, Institut für Informatik, Universität Stuttgart 2000*

[Ram99] Magnus Ramage, Carole BrookeGold, Malcolm Munro, Keith Bennett, Nicolas Gold, "Overcoming The Legacy Dilemma: Modelling Socio-Technical Change Options", *in Proceedings of Systems Modelling for Business Process Improvement*, Coleraine, 29-31 March 1999.

[Ric02] T.Richner, S.Ducasse. "Using dynamic information for the iterative recovery of collaborations and roles", *Proceedings of International Conference of Software Maintenance, IEEE Computer Society,* 2002, pp 34-43.

[Ric97] R. Richardson, D. Lawless, J. Bisbal, B. Wu, J. Grimson, and V. Wade "A Survey of Research into Legacy System Migration", *Technical Report TCD-CS-1997-01, Computer Science Department, Trinity College Dublin.* January 1997.

[Riv00] C. Riva. "Reverse architecting: An industrial experience report". *In Proceedings 7th Working Conference on Reverse Engineering (WCRE)*, pages 42–51. IEEE Computer Society, 2000.

[Rug95] S. Rugaber, "Program Comprehension." *In the Encyclopedia of Computer Science and Technology, 35(20),* Marcel Dekker, Inc:New York, 341-368,1995

[Raj00] V. Rajlich, K. Bennett: "A new perspective on software evolution: the staged model", IEEE Computer, July 2000, 66-71.

[Rou04] Mohamed H. Rouane, Petko Valtchev and Houari Sahraoui and Marianne Huchard, "Merging conceptual hierarchies using concept lattices", *In the MASPEGHI Workshop, MechAnisms for SPEcialization, Generalization and inheritance at ECOOP 2004*, June, 2004

[Rus02] Jeffry Russel. "Program slicing for codesign." *In Tenth International Symposium on Hardware/Software Codesign*, 2002.

[Sah02] H. Sahraoui, P. Valtchev, I. Konkobo, S. Shen, "Object Identification in Legacy Code as a Grouping Problem", *In the proc. of the 26th Computer Software and Applications Conference (COMPSAC'02)*, Oxford, 2002.

[Sah99] Houari A. Sahraoui, Hakim Lounis, Walcelio Melo, Hafedh Mili, "A Concept Formation Based Approach to Object Identification in Procedural Code", *Automated Software Engineering Journal* (1999).

[Sah97] Sahraoui H. A., Melo W. L., Lounis H., Dumont F., "Applying concept formation methods to object identification in procedural code". *In the proc. of the IEEE Automated software engineering conference (ASE'97)*, 1997.

[Sch91] Schwanke, R.W, "An intelligent tool for re-engineering software modularity", *International Conference on Software Engineering,* pp. 83-92, May 1991

[Sco98] T. Scott, "A Reverse Engineering Environment Framework", *Technical Report, CMU/SEI-98-TR-005, Carnegie Mellon Software Engineering Institute,* Pittsburgh, April 1998

[She03] Shiqiang Shen, "Object Identificaiton Using Conceptual Clustering", *Master degree thesis*, University of Montreal, Computer Science Department, 2003

[Sif97] M. Siff ,T. Reps. "Identifying modules via concept analysis". *In International Conference on Software Maintenance,* ICSM97. IEEE Computer Society, 1997.

[Som00] Ian Sommerville "Software Engineering", *6th edition. Addison-Wesley 2000*

[Sne95] H. M. Sneed and E. Ny´ary. "Extracting object-oriented specification from procedurally oriented programs". *In Second Working Conference on Revers Engineering;* WCRE'95, pages 217–226. IEEE Computer Society, 1995.

[Sne98] H. M. Sneed and R. Majnar, "A Case Study in Software Wrapping", *in Proc. of ICSM'98, IEEE Computer Society, 1998, pp. 86-93.*

[Sne00] G. Snelting. "Software reengineering based on concept lattices". *In Proceedings of the 4th European Conference on Software Maintenance and Reengineering,* (CSMR'00). IEEE Computer Society, 2000.

[Sta00]ANSI/IEEE Standard, "Recommended Practice for Architectural Description of Software-Intensive Systems", 1471-2000.

[Sta98] John T. Stasko, John B. Domingue, Marc H. Brown and Blaine A. Price. "*Software visualization: programming as a multimedia experience*", MIT Press, 1998.

[Ste98] P. Stevens and R. Pooley. Systems reengineering patterns. In *ACM SIGSOFT Foundations of Software Engineering (FSE-98),* Lake Buena Vista, Florida, USA, pages 17–23. ACM Press, 1998.

[Sto00] M.-A Storey, K. Wong and H.A. Müller. "How Do Program Understanding Tools Affect How Programmers Understand Programs?" *Journal of Science of Computer Programming*, Vol. 36, Nos. 2-3, pp. 183-207, March 2000.

[Sto03] C. Stoermer, L. O'Brien, C. Verhoef, "Moving Towards Quality Attribute Driven Software Architecture Reconstruction", *Working Conference on Reverse Engineering, Victoria, BC, Canada,* November 13th - 16th, 2003

[Sys99] T. Systa. On the relationships between static and dynamic models in reverse engineering java software. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE-99),* Atlanta, Georgia, USA, pages 304–313. IEEE Computer Society Press, October 1999.

[Sys01] T. Systa, K. Koskimies, H. Muller. "Shimba – an environment for reverse engineering java software systems." *Software –Practice and Experience,* 1(1), January 2001.

[Tak01] T. Takada, F. Ohata, and K. Inoue. Dependence-cache slicing: A program slicing method using lightweight dynamic information. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 169-177, 2001.

[Tan98]W.-G TAN and G.G. Gable, "Attitudes of Maintenance Personnel Towards Maintenance Work: a Comparative Analysis", *Journal of Software Maintenance: Research and Practice. January 1998*

[Tea00] S. Teasley, L Covi, M.S. Krishnan. J.S. Olson, "How Does Radical Collocation Help a Team Succeed?", *Computer Human Interface Proceedings, 2000.*

[Til96] S. Tilley, S. Paul, and D. B. Smith; "Towards a Framework for Program Understanding." In *Proceedings of the International Workshop on Programming Comprehension (IWPC 1996). IEEE Computer Society, 1996*

[Til98] Scott Tilley. "A Reverse Engineering Environment Framework", *Technical Report of Carnegie Mellon Software Engineering Institute,* Pittsburg, PA, April 1998.

[Tip95]Frank Tip, "A survey of program slicing techniques." *Journal of programming languages*, 3(3), September 1995.

[Uma97] A. Umar. "*Application (Re)Engineering: BuildingWeb-Based Applications and Dealing with Legacies.*" Prentice Hall, 1997.

[Val01] P.Valtchev, R. Missaoui, "Building Galois (Concept) Lattices from Parts: Generalizing the Incremental Approac". *Proceedings of the ICCS 2001,* LNCS 2120, Springer Verlag, pp. 290-303, 2001

[Val03] P. Valtchev, M. H. Rouane, M. Huchard, and C. Roume, "Extracting Formal Concepts out of Relational Data," *in Proceedings of the 4th Intl. Conference Journ'ees de l'Informatique Messine (JIM'03): Knowledge Discovery and Discrete Mathematics,* Metz (FR), 3-6 September, E. SanJuan, A. Berry, A. Sigayret, and A. Napoli, eds., pp. 37–49, INRIA, 2003.

[Van00] A. van Deursen and L. Moonen. "Exploring Legacy Systems Using Types". In *Proceedings 7th Working Conference on Reverse Engineering, (*WCRE'2000*)*, pages 32-41. IEEE Computer Society, 2000.

[Wal98] R.J.Walker, G.C.Murphy, B.F.Benson, D.Wright, D.Swanson and J.Issaak. "Visualizing dynamic software system information through high-level models", *Proceeding OOPSLA'98, 1998, pp.271-283.*

[Wmc01] "*The Workflow Management Coalition Specification*", Workflow Management coalition Work Group, Oct.2001

[Wu05a] L.Wu, H. Sahraoui, P. Valtchev, "Coping with Legacy System Migration Complexity", *in the proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05),* IEEE Computer Society Press, Shanghai, China, June 2005

[Wu05b] L.Wu, H. Sahraoui, P. Valtchev, "Caribou: a Supporting Environment for Software e-Development", *in the Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05),* IEEE Computer Society Press, Hongkong, February 2005

[Wu04a] L.Wu, H. Sahraoui, P. Valtchev, "Automatic Detecting Code Cooperation", *in the Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04),* IEEE Computer Society Press, Seoul, Korea, November 2004

[Wu04b] L.Wu, H. Sahraoui, "Supporting Web Collaboration for Cooperative Software Development", *in the Proceedings of the 2004 IEEE/ACM International Conference on Web Intelligence(WI'04),* Beijing, China, September 2004

[Wu04c] L.Wu, H. Sahraoui, P. Valtchev, "Program Comprehension with Dynamic Recovery of Code Collaboration Patterns and Roles", *in the Proceedings of 14th IBM Centers for Advanced Studies Conference (CASCON'04)*, Toronto, Canada, October 2004

[Wu03a] L.Wu, H. Sahraoui, P. Valtchev, "Legacy Design Recovery with Dynamic Visualization", i*n the Proceedings of the 16th International Conference Software & Systems Engineering and their Applications (ICSSEA'03),* Paris, France, December 2003

[Wu03b] L.Wu, H. Sahraoui, P. Valtchev, "Migrating Legacy Software towards new Technologies", i*n the Proceedings of the Migration and Evolvability of Long-life Software Systems Workshop,* NetObjectDays, Erfurt, Germany, September 2003

[Yan03] H. Yang and M. Ward, "Succesful Evolution of Software Systems", Artech House, ISBN 158-053349-3,2003

[Zel03] A. Zeller: "Program Analysis: A Hierarchy". Proc. *Workshop on Dynamic Analysis (WODA 2003)*, Portland, Oregon, May 2003.

[Zha98] J. Zhang, "A Distributed Representation Approach to Group Problem Solving", *Journal of the American Society for Information Science,* Volume 49, Number 9, 1998, pp. 801-809.

[Zou01] Ying Zou, Kostas Kontogiannis, " A Framework for Migrating Procedural Code to Object-Oriented Platforms", T*he proceedings of 8th Asia-Pacific Software Engineering Conference, Macau SAR, China, December 4-7, 2001.*