THE REUSE-ORIENTED APPROACH AND ITS CASE STUDY

by
Zhiyi Lou

School of Computer Science
McGill University, Montreal

November 1993

ABSTRACT

Reuse is widely believed to be a key to achieving the dramatic improvement in productivity and quality the software industry requires. Although experience shows that certain kinds of reuse can be successful, general success has been elusive. From the technical aspect, three kinds of problems inhibit the advance of reusable software engineering: organizational problems, representational problems and operational problems. Our study aims at removing these barriers by introducing a reuse-oriented approach to software development in general and systematic reuse in particular. On the basis of the object-oriented methodology, this approach presents an incremental development paradigm that coordinates the interaction between development process and reuse process with two parallel organizations, and incorporates four technical issues that support the development paradigm: broad-spectrum reuse, domain-oriented software life cycle, multi-organization development process model and experience factory. The final product is a reuse-enabling software system within an application domain. The reuse-oriented approach is demonstrated with the RECPAM system, a statistical application that includes a family of projects. The development of the RECPAM system is arranged in two steps: the creation of the reuse-enabling system and the development of the reuse-enabling system. In the first step, we focus on developing a general RECPAM system for reuse to demonstrate how a starter reusable system can be achieved. In the second step, we focus on developing concrete projects with reuse to illustrate how development can be improved by applying the general system to different sorts of projects within the RECPAM domain.

RÉSUMÉ

La réutilisation est considérée comme essentielle pour améliorer la productivité et la qualité, primordiales dans l'industrie du logiciel. Bien que l'experience a montré que certains types de réutilisation peuvent bien réussir, une réussite généralisée, but ultime n'a pas encore été obtenue. Les techniques de logiciels réutilisables font face à trois sortes de problèmes majeurs qui sont: les problèmes d'organisation, de représentation et d'opération. Le but de notre étude est de parvenir à surmonter ces obstacles en adoptant une approche orientée vers la réutilisation de logiciels en général et une réutilisation systématique dans des cas particuliers. Cette approche de réutilisation de logiciels présente un exemple de méthode améliorée qui coordonne l'interaction entre le processus de développement et celui de réutilisation avec deux organisations parallèles, tout en incluant quatre aspects techniques: réutilisation à grande echelle, cycle de vie du logiciel orienté-domaine, modèle de développement d'un processus à organisation multiple et usine d'experience. Le produit final est un système susceptible d'être utilisé dans le cadre d'un domaine d'application donné. L'approche orientée vers la réutilisation est mise en œuvre dans le développement du système RECPAM, une application statistique incluant une famille de projets similaires. Le développement du système RECPAM a été effectué en deux étapes: création d'une base abstraite rendant possible la réutilisation et développement dún système concret réutilisable. La première étape est consacrée au développement général du système RECPAM avec pour objectif de rendre le processus de réutilisation de logiciel plus facile. La seconde étape est consacrée au développement de projets de réutilisations spécifiques, dans le but de prouver que le développement peut être grandement facilité par l'application du système général à différentes sortes de projets dans le domaine RECPAM.

ACKNOWLEDGMENTS

I am greatly indebted to Dr. M. Newborn for his valuable advice, encouragement and assistance during the course of this study.

I wish to express my sincere appreciation to Dr. A. Ciampi for his guidance, consistent support and constructive suggestion, and for providing financial support from his MRC (the Medical Research Council of Canada) research grant. I acknowledge the Montreal Children's Hospital Research Institute for supplying a conducive environment.

Many thanks are extended to Mr. A. Negassa, Ms. L. Hendricks and Mr. Q. Yu, for their technical assistance and participation during the development and testing of the RECPAM system. My special thanks also go to Prof. A. Godfrey for his enthusiastic help in preparing this manuscript.

Finally, my deepest gratitude is extended to my dear wife Tian for her love, patience and understanding, to my lovely daughter Meng, and to my parents and parents-in-law for their moral support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1. INTRODUCTION

1.1 Software Reuse Concept

As the demand for software continues to grow, software developers are searching for ways to develop software more quickly and efficiently. One of them is to reuse software already written. Reuse is a general engineering principle whose importance derives from the desire to avoid duplication and to capture commonality among inherently similar situations. It provides both an intellectual justification for research that simplifies and unifies our understanding of phenomena and an economic justification for production that increases productivity and quality [Wegner 1984]. In well-established disciplines, like civil engineering or electrical engineering, reusability is a standard part of development. Electrical engineers, for example, consult component catalogs continuously during the design process to check what available part best fits the design constraints. In many cases, the original design requirements are modified to take advantage of existing components. However, it has for a long time been recognized that one fundamental weakness of the software industry is the fact that new software systems are usually constructed "from scratch" [Horowitz and Munson 1984]. During initial software development, reuse may be totally absent, but more often it manifests itself as the informal reuse of in-head knowledge about older, similar systems. In most cases, reuse is merely the sharing of a set of routines in a run-time library that is designed to be common to both existing and planned systems. During maintenance, reuse is sometimes formalized through a number of project specific support tools. Why isn't software more like hardware? Why must every new development start from scratch? There should be catalogs of software modules as there are catalogs of

VLSI devices: When we build a new system, we should be ordering modules from these catalogs and assembling them, rather than reinventing the wheel every time.

Software reuse can be extensively defined as the reapplication of a variety of kinds of software-related materials about one system to another similar system in order to reduce the effort of development and maintenance of that other system. The reused materials include all forms of artifacts during software development such as domain knowledge, development experience, design decisions, architectural structures, requirements, designs, codes, documentation, and so forth. This expensive definition breaks the limitations of traditional views of reuse which center on the reapplication of code components. It results in the ampler exploitation of software reuse and thereby earns the maximum benefits on its investment. The software reuse will amplify the programming capabilities, reduce the amount of work needed on new systems, and achieve a better overall control over the production process and the quality of its products. The benefits offered by successful software reuse can be summarized as follows:

- Productivity - Use existing components. Increased reuse helps reduce the efforts needed to develop software systems.
- Reliability - Use proven components. Developing reliable software is difficult, especially for large, complex systems. Software reuse helps by providing components whose reliability is already demonstrated.
- Consistency - Use the same components in many places. Through a set of widely useful components, software reuse helps reduce the need for fresh, and possibly idiosyncratic, design.
- Manageability - Use well-understood components. Increased reuse helps lessen the likelihood of cost and schedule overruns by providing already developed components whose behavior is understood.
- Standardization - Use standard components. With reuse, software components are in place early to help users and developers with specification and implementation.

- Knowledge capture - Use encapsulated components. The application domain expertise and development process knowledge hidden in components are easily captured with reuse process.

Reusing software is a simple and straight-forward concept that is behind almost every software development. It manifests itself in many forms. The following popular mechanisms can be considered simple examples: utilizing a set of library routines; connecting programs with a UNIX pipe; utilizing operating system services calls; and salvaging components from previously written systems.

1.2 Software Reuse Review

The concept of software reuse is not new. It has been part of the programming heritage since the origins of the stored program computer EDSAC at the University of Cambridge in 1949. Maurice Wilkes first recognized the need for avoiding redundant effort in writing scientific subroutines, and recommended a library of routines be kept for general use [Tracz 1987]. A specialized form of software reuse, libraries of standard functions, has been in widespread use. Various technical developments in this area are all relevant to reuse of code but were not developed with reuse particularly in mind.

In 1969, McIlroy reformulated this concept and proposed the idea of a software components catalog from which software parts could be assembled, much as is done with mechanical and electronic components [McIlroy 1969]. Programmers began efforts to introduce reusability into their software development processes at this time. In the late 70s, this idea was applied in a limited domain by Lanergan and Poynton with excellent results. They identified and classified a lot of code and standard structures that could be used in many of their applications [Lanergan and Poynton 1979]. Since then, Japanese software factories have reported great improvements in programmer productivity through reusability by integrating known techniques from different disciplines like resource management, production engineering, quality control, software engineering, and industrial

psychology [Matsumoto 1981] [Kim 1983] [Tajima and Matsubara 1984]. Software development started concerning both intentionally producing reusable software components and explicitly utilizing them.

Interest in reuse burst onto the software scene in 1983 with the landmark ITT Workshop on Reusability in Programming. Biggerstaff and Perilis succeeded in attracting a critical mass of leading researchers and industry representatives to this meeting. A varieties of approaches were proposed, and more and more technologies emerged. Massive research on reusability focused on methods and mechanisms to perform reuse, on the presentation of reusable components, and on the organization of repositories of components. Freeman introduced software reuse as a topic of software engineering research, so that the construction of software became an engineering task [Freeman 1983]. He emphasized the reuse of all information generated during the software development process and proposed a set of long-term research directions relevant to effective software reuse. Matsumoto formally proposed the *software factory*, in which paradigms of industrial production are adopted for software production, in terms of software reusability [Matsumoto, *et al.* 1981] [Matsumoto 1984]. Weger suggested that software production is a capital-intensive process. The reusable software is seen as capital goods whose development cost may be recovered from its set of uses. Thus, technologies which identify capital goods with reusable resources, and capital with reusability, are becoming more powerful and expensive, and it requires greater early investment to reduce later expenditures [Weger 1984].

More recently, Basili introduced a systematic reuse approach for supporting comprehensive reuse. He presented a comprehensive framework for reuse consisting of a reuse model, model-based characterization schemes, the TAME environment model supporting the integration of reuse into software development, and ongoing research and development efforts toward a TAME environment prototype [Basili, *et al.* 1991] [Basili and Rombach 1988]. Today, many organizations view software reusability as an

indispensable technology that must be developed to ensure future competitiveness [Prieto-Diaz and Jones 1988].

There is a great variety of possible approaches to software reusability. The two primary groups are the composition-based approach and the generation-based approach. The differences between then depend on the nature of the components being reused and on the technologies applied to the reuse process [Biggerstaff and Richter 1987] [Biggerstaff and Perlis 1984].

Composition-based approach: The reusable components, which are called *building blocks*, are largely atomic, almost immutable and passive elements operated on by external agent. They are reused through composition.

Reusable building blocks can span all levels of software, including specification, design, and code. They may range in size from complete subsystems down to individual modules or fragments. Both application-specific components and general purpose components are important. Some components can be reused "as-is" while others may have to be customized for each application [Lenz, *et al.* 1987] [Burton, *et al.* 1987] [Korson 1992] [Lubars 1987]. There are two typical approaches currently used in this group: the standard component approach and the common utility approach.

(1) Standard component approach

The standard component approach standardizes the application-dependent function groups and provides them as standard components. The programmer develops his software while combining these components and creating functions that are lacking.

(2) Common utility approach

The common utility approach aims only at standardizing low-level functions and providing them as common utilities.

There are many important issues — technical and managerial — related to implementing a successful reuse in this group. We discuss only two of them, software factory and domain analysis, because they are directly invoked in our approach.

(i) *Software factory*

The software factory concept was originally proposed to improve software development productivity through standardized tools, methods and component reuse. As the earliest proponent, Bemer gave the first working definition of what might constitute a software factory: "A software factory should be a programming environment residing upon and controlled by a computer. Program construction, checkout, and usage should be done entirely within this environment and by using the tools contained in the environment" [Bemer 1969]. He focused on standardized tools and controls. McIlroy emphasized another factory-like concept: systematic reusability of code when constructing new programs. He used the term "factory" in the context of facilities dedicated to producing parameterized families of software parts or routines that would serve as building blocks for tailored programs reusable across different computers [McIlroy 1969]. The first company in the world to adopt the term "factory" to label a software facility was Hitachi, which founded the Hitachi Software Works in 1969. By the late 1960s, the term" factory" had arrived in software engineering considering more efficient software development approaches. This label became especially popular in Japan during mid-1970 and 1980s [Cusumano 1989]. In order to support a comprehensive framework for the reuse he proposed, Basili defined the term "experience factory" as a logical or physical organization that supports project development by analyzing all kinds of experience, acting as a repository for such experience, and supplying that experience to various projects on demand. His experience factory can be divided into two independent sub-organizations: domain factory and component factory. Domain factory defines the process for producing applications within the domain, implements the environment needed to support that process, and monitors and improves that environment and process. The component supplies factory software components to projects upon demand, and creates and maintains a repository of chosen components for future use. Basili presented the architecture of the component factory at three levels of abstraction: reference level,

conceptual level and implementation level, and defined a reference architecture from which specific architectures can be derived by instantiation [Basili, *et al.* 1991].

(ii) *Domain Analysis*

Domain analysis was first introduced by Neighbors as "the activity of identifying the objects and operations of a class of similar systems in a particular problem domain" [Neighbors 1980]. He draws the analogy of domain analysis to system analysis. The difference is that system analysis is concerned with specific actions in a specific system, while domain analysis is concerned with actions and objects in all systems in an application area. During his research with Draco, a code generator system that works by integrating reusable components, he pointed out "the key to reusable software is captured in domain analysis in that it stresses the reusability of analysis and design, not code". The Common Ada Missile Packages (CAMP) Project took Neighbors' ideas into practice. The CAMP Project is the first explicitly reported domain analysis experience [CAMP 1987]. McCain makes an initial attempt at addressing this issue by integrating the concept of domain analysis into the software development process. He proposes a "conventional product development model" as the basis for a methodology to construct reusable components [McCain 1985]. There are many approaches. Prieto-Diaz proposed a more cohesive procedural model for domain analysis. He extended the methodology for deriving specialized classification schemes in library science to domain analysis as a procedural model in a series of data flow diagrams. He defined specific activities and intermediate products. A project at GTE Laboratories is currently underway using this model [Prieto-Diaz 1987]. Shlaer presented an object-oriented approach to domain analysis that is fundamentally based on objects. His approach is based on building three types of formal models: an information model, a set of state models, and a set of process models and boundary statement [Shlaer and Mellor 1989]. Arango focused on this concern and proposed a different approach to domain analysis. The basic premise in this approach is to see reuse as a learning system. The software development process is seen as a self-

improving system that draws from a reuse infrastructure" as the knowledge source. Domain analysis is then a continuing process of creating and maintaining the reuse infrastructure[Arango 1988, 1989].

Generation-based approach: The reusable components, which are called *patterns*, are diffuse, malleable and active. They are often patterns woven into the fabric of a generator program. The reusability is achieved by program generators. It is a natural extension of the composition-based approach and has the potential of much greater payoff. We distinguish three subclasses of generation-based approaches based on the properties of reusable patterns that are emphasized: the language-based approach, the application generators approach, and the transformation system approach.

(1) Language-based approach

Language-based generation approach is an approach in which the specification language is well defined, truly represents a problem domain, and hides the details of implementation from its user. Reuse is enhanced by such language specifically because it does hide the details of implementation and raises the level of discourse to the application domain level rather than the implementation level. The reusable patterns are integrated with the compiler of a specific language. Language-based approach includes very high-level languages and problem-oriented languages.

The paper by Dubinsky *et al.* describes using the SETL, a language based on the notion of representing computations as operations on mathematical sets, to specify a large program, and then transforming that specification into efficient implementation. For many problems, this significantly simplifies the expression of the computation, although it often makes the generation of efficient code a challenge [Dubinsky, *et al.* 1989].

(2) Application generator approach

Application generators embed in their design the architectural pattern that will be reused in the course of generating specific instances of target systems. Thus the instances generated have that architectural pattern in common.

Darco system is typical of this class. Draco could be put in any one of the three generation categories. It requires the development of a domain-specific language in which the user can specify his or her problem, it generates target programs from domain-specific specifications, and it uses a set of user-defined transformations to accomplish this generation [Neighbors 1989, 1984].

(3) Transformation-based approach

The transformation-based approach focuses upon the role, structure, and operation of transformations in the evolution of high-level specifications into operational programs. Transformation systems are based on the idea of describing the target system in an easy to understand, easy-to-use language, and then refining it into an executable, efficient target program. The reusable patterns are most often embedded into transformation rules.

Arango *et al.* have used a transformational reusability support system to port the system itself into another target environment - they claim that the approach is a very powerful transformation-based maintenance model that allows an undocumented source program to be ported without any modifications into another environment, where it can be reused [Arango, *et al.* 1986]. Boyle and Muralidharan present a system transforming pure Lisp programs into Fortran code, where the Lisp program is seen as an abstract specification for the Fortran version. Transformation rules include many reusable patterns for LISP to FORTRAN translations, but no broader reusable information for the software development process [Boyle and Muralidharan 1984]. Cheatham, on the other hand, suggests transformation systems for a software engineering paradigm. An environment supporting the methodology that facilitates the reuse of abstract programs written in a domain-dependent language, which is extended from a base language, has been developed by his group. The abstract programs are transformed into their concrete counterparts by using transformation rules [Cheatham 1984].

Unfortunately, over the broad span of systems, reuse is exploited only to a very limited extent today. Although experience shows that certain kinds of reuse can be successful,

general success has been elusive. Reuse still is a great promise which has been largely unfulfilled [Biggerstaff and Richter 1987] [Tracz 1987, 1988] [Basili and Rombach 1991].

From a nontechnical perspective, Meyer identified the following reasons: "(i) Economic incentives tend to work against reusability. If you, as a contractor, deliver software that is too general and too reusable, you won't get the next job—your client won't need a next job! (ii) The famous not-invented-here complex also works against reusability. (iii) Reusable software must be retrievable, which means we need libraries of reusable modules and good database-searching tools so client programmers can find appropriate modules easily." [Meyer 1987]

From a technical perspective, this is due to the difficulties both in implementing true production environments for reusable modules that could successfully support classification, storage and retrieval of reusable components [Prieto-Diaz 1985] and in constructing production-quality versions of new software engineering paradigms that support active reusable patterns of the production process rather than passive reusable building blocks [Neighbors 1980]. More illustrations will be given in the next chapter.

CHAPTER 2. THE REUSE-ORIENTED APPROACH

2.1 Motivations

The existing gap between demand and our ability to produce high-quality software at a high level of productivity and in a short period of time cost-effectively calls for the evolution of modern software development methodologies. A reuse-based software development approach could fill the gap by synthesizing the three goals: improve the effectiveness of the process, reduce the amount of rework, and reuse life cycle products. This approach encourages systematically adopting an incremental development style which provides opportunities for economies in software development. It has been observed that reuse has been practiced in software development for many decades and is behind every software system. Unfortunately, it is insufficiently taken into account in most software development methodology. Although experience shows that certain kinds of reuse can be successful, general success still has been elusive [Basili and Rombach 1991]. Perhaps we reuse unconsciously, informally, and inefficiently. Is it possible to assume that any new software development is first based on reusing all kinds of software-related efforts from prior developments and then offers its own current efforts to be reused in other system developments? The primary goal of our research is to derive a software development approach, that minimizes the amount of each new system that has to be developed from scratch, by systematically employing reuse as a major strategy of improving the development process. Based on the implicit inheritance of software development processes and the natural sharing of all kinds of software-related artifacts between similar systems, a reuse intensive software development approach can be evolved by effectively and efficiently integrating the software reuse process into a convenient software development

process. Reuse will be the key to enabling the evolutional approach to achieve the dramatic improvement in productivity and quality required to satisfy anticipated growing demands. Quality should improve by reusing all forms of proven experience including products, processes, as well as quality and productivity models. Productivity should increase by using existing experience rather than creating everything from scratch. From the following considerations regarding software development in general and reuse in particular, we propose the reuse-oriented software development approach to guide the process of developing a family of similar systems within an application domain.

First, reusing is a key ingredient to progress in any discipline. Without reuse everything must be relearned and recreated; progress in an economical fashion is unlikely. Reuse is less institutionalized in software engineering than in other engineering disciplines because of the following unsolved technical problems in the software industry described below. As reuse intensive process, the reuse-oriented approach is intended to resolve them in order to make reuse more attractive in software development.

Organizational problem

Most reuse occurs in an ad-hoc fashion rather than as result of planning and support. Present software systems are often not initially designed for future reuse. A project's focus is system delivery. Packaging software-related experience for reuse is at best a secondary concern. Therefore, it is rarely feasible to decompose an existing software system into reusable modules that can be then used to construct other similar systems or to formalize specific system development process in reusable forms. Also, existing process models, which tend to be rigidly deterministic, are not defined to take advantage of reuse, much less to create reusable artifacts. In order to achieve effective and efficient reuse, reusability must be engineered from the start, and be treated as an integral part of system development rather than an afterthought of the implementation. This requires that a software development process deal with two goals concurrently: how to produce software-related resources with maximum potential for reuse (development for reuse) and

how to develop new systems making the most effective use of these resources (development with reuse). Each goal results in different major concerns to be emphasized and different process models to be supported, but the two goals are highly correlated in some content, such as operated objects, time sequence and so on. The problem is how to unify two different goals in the same process. A multiorganization framework will provide the best solution.

Representational problem

There exists a wide gap between the kinds and forms of knowledge available about problem domains or development processes and the content and form of the artifacts that can be reused in software construction. For instance, knowledge about an application domain or development is often implicit and nonformal, while reusable artifacts must usually be represented explicitly and formally. The real knowledge is normally a contextual and complex entity, while the reusable artifacts must be recorded as context-free and factored modules. The reuse process involves two transformations with opposite directions: a software development is packaged as a collection of reusable artifacts, and a collection of reusable artifacts is reused as the basic units of new software development. The problem is how to define an appropriate representation which supports both transformations: packaging reusable objects (including identify, extract, record and catalog) and reusing reusable artifacts (including recognize, retrieve, customize and compose). Moreover, the representation must at least reveal the reusability with the higher pontential payoff, the generality for a broader range of applications, the cheaper modification for transferability and less integration effort. There are a number of dilemmas among these requirements [Biggerstaff and Richter 1987].

Operational problem

Software reuse is not a specific technique, algorithm, heuristic or set of guidelines. It is many different mixtures of technologies, process modules and cultures. This demands a radical departure from the operational styles prevalent in current programming. Software

reuse involves many operations and is applied to various phases across the development process, and reusable objects are capable of capturing all kinds of software-related information. However, much current work tends to focus on a few phases in the development process, and on a particular phase without addressing the transition and traceability between phases. Most existing systems are limited to only reuse resource code, the lowest level of reusable object. Synthesis of a variety of technologies, process modules and cultures is lacking. The problem is how to comprehensively perform each operation of reuse process and how to systematically blend the reuse process with the development process to make reuse cost-effective.

Secondly, software reuse comes in many flavors and does not, by itself, provide a comprehensive approach to software development. For that reason, the reuse-oriented approach attempts to center around the reuse and incorporate other supporting technical issues to propose a systematic development paradigm and guidelines, which will be combined with an appropriate method to derive a practical reuse-based methodology. As a major means to improve system development process, the reuse-oriented approach seeks to make the reuse:

- More systematic, across various phases of a system development and across various project developments within an application domain.
- More comprehensive, mixing many different technologies, process models and cultures in appropriate phases of development process for fostering reuse.
- More dynamic, with new reusable experience accumulating over time as a by-product of project development, and continuously refining existing experience as the feedback of reuse process. It presents a truly incremental development environment.
- More extensive, encompassing not only code and design but also specifications, analysis, knowledge, testing and so on. It is expected to reuse all kinds of software-related artifacts.

Finally, the object-oriented methodology, which includes object-oriented analysis, object-oriented design and object-oriented programming, is considered the best one for reuse because it fosters the successful reuse of multiple aspects. Essential to the object-oriented methodology is the view of "objects", which are encapsulated units, akin to modules, and which permit the abstraction of real-world entities into software terms. This leads to the perspicuity of the representation and its tendency to promote larger and more abstract reusable objects [Frakes *et al.* 1991]. Object orientation has a high degree of continuity from one phase of the life cycle to the next. It allows the integration of the various phases of software development within a single framework using common concepts and (often) notations. This opens the way to make reuse possible at all phases, such as reuse of design models, or architectures [Wirfs-Brock and Johnson 1990], and even analysis models from relevant problem domain [Champeaux and Faure 1992]. In addition, object-oriented development introduces a number of advanced techniques and mechanisms for emphasizing and facilitating reuse operations. For example, encapsulation, inheritance, polymorphism, and dynamic binding certainly overcome many technical barriers to large scale reuse. For these reasons we invoke object-oriented methodology as the conceptual foundation for proposing the new approach.

2.2 Development Process and Reuse Process

Reuse, the fundamental goal of our research, will be explicitly addressed by integrating the reuse process into the software development process due to the following two facts. At each phase in the development process, we should be considering how previously completed work can be used to reduce the effort needed for the current task. It is obvious that the reuse process is neither an additional phase, nor an alternative to any one phase, nor a part of any one phase in the development process. And we do not want to limit consideration of reuse to any one specific phase of the development process. The reuse of an object from some earlier phase will probably cause the reuse on a large scale of objects

1981] [Bauer 1982] [Cheatham 1984]. For instance, in the operational-transformational model, software development proceeds from an executable problem-oriented specification through a sequence of transformations to a more efficient implementation-oriented realization. Early phases are independent of computational resources. Transformations from the problem-oriented specification to an efficient implementation are automatic wherever possible. Maintenance and enhancement changes are performed on the problem-oriented specification, which is then optimized.

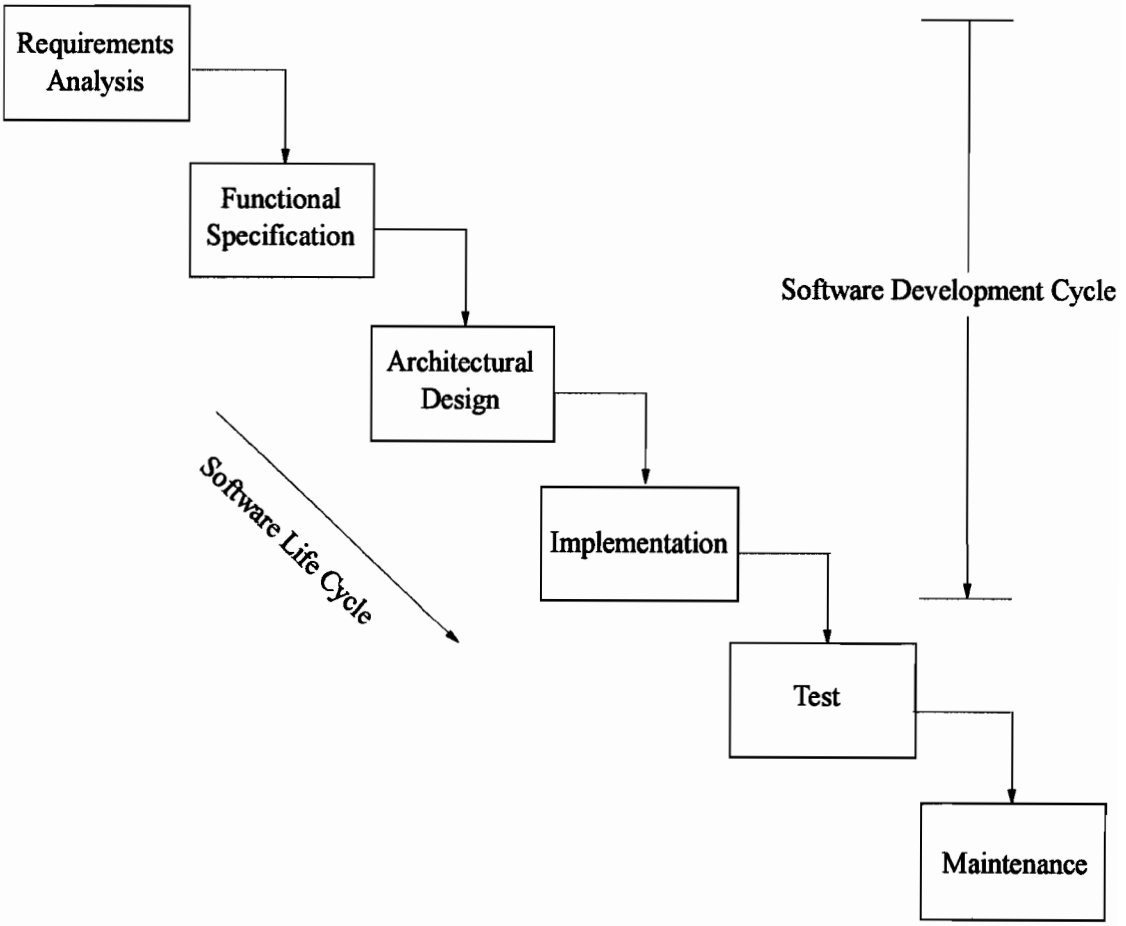

Figure 1. Software life cycle

These models are not at all independent. They each derive from systems engineering research in the 1960s. Each model type introduces different notations of phases through which sequenced progress is made in the development of the final deliverable product. The phases have been modified from their initial systems engineering origins to serve the particular needs of the software development process. The names, boundaries and order of progression may differ from model to model. However, each model follows a progression from requirements through analysis, function specification, architectural design, implementation, test, and maintenance. It can be formalized to six qualitatively different phases in linear sequence. Certain tasks are assigned to each phase in the life cycle. A model allows effective division of the work involved in developing the system. Figure 1 is a high-level view of software life cycle which can be specialized to any particular life cycle model. We only focus on the front portion of the software life cycle: requirement analysis, functional specification, architectural design, and implementation, and refer to the first four phases as *software development cycle*.

(1) *Requirements analysis*: Requirements analysis is the process of determining and documenting the customer's purposes and the constraints on its development. It can be viewed as the design of a set of goals for the proposed system.

(2) *Functional specification*: Functional specification is the process of developing and formalizing a proposed systems interface for meeting the customer's needs. It can be viewed as the design of external interfaces.

(3) *Architectural design*: Architectural design is the process of decomposing the system into modules and defining internal interfaces. It can be viewed as the design of internal interfaces.

(4) *Implementation*: Implementation is the process of coding a program that correctly realizes the specified interface for each module identified during architectural design, and that meets the associated performance requirements. It consists of three main activities: choosing data structures and algorithms, working out the details of the

code, and checking the correspondence between implementation and the specified interface. It can be view as the design of data structures and algorithms.

2.2.2 Software Reuse Process

Complete software reuse involves two opposite directions of transformation processes: learning and reusing. The learning process is generalizing software-related experience extracted from a particular system development in reusable form. In contrast, the reusing process is specializing the general reusable forms to adapt it to new system development. In a similar way, we can use the concept of the life cycle to organize and manage the reuse process. Certain operations are performed in each phase in the reuse life cycle [Boldyreff 1989]. A reuse life cycle can be divided into three separate phases along a time dimension. They are pre-reuse, reuse and post-reuse, shown in Figure 2.

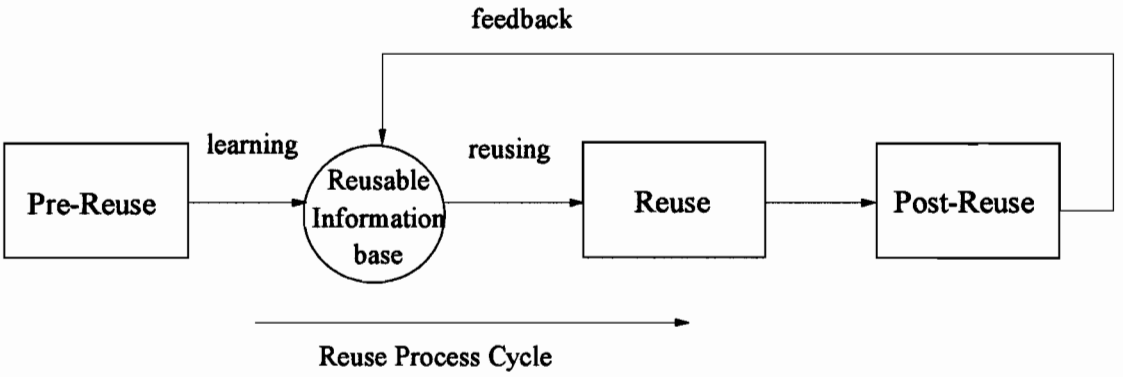

Figure 2. Software reuse life cycle

(1) *Pre-reuse*: Pre-reuse is the process of acquiring all kinds of software-related experience from any software development process in reusable form. It can be viewed as learning new experience. This phase can be divided into four activities, shown as Figure 3(a):

- Identification: identifying new potentially reusable objects from any one application development process.
- Extraction: extracting these identified objects from their context of development and from the context of the application system.
- Recording: representing these extracted objects as reusable modules in generalized descriptive forms.
- Cataloging: classifying and cataloging the large collections of reusable modules in a readily available way.

(2) *Reuse*: Reuse is the process of identifying the appropriate experience from a experience base and customizing it to fit the given specific requirements. It can be viewed as iteratively reusing prior experience. This phase also comprises four activitiess, shown as Figure 3(b).

- Recognition: recognizing what parts of the current system can use previously existing reusable objects.
- Finding: searching the best match reusable candidate from the large collection of reusable modules according to the outcome of recognition operation.
- Customizing: making the reusable candidate to fit the specific requirements of the new application development by modification.
- Integration: embedding the customized object into the context of the current development process and the context of new application system.

(3) *Post-reuse*: Post-reuse is the process of feeding back any necessary refinement to existing experience. It can be viewed as updating existing experience. This is the special form of learning, shown as Figure 3(c).

- Evaluation: after reusing the existing experience, determining whether it needs improvement in quality or reusablility.
- Updating: improving the existing reusable objects according to previous evaluation.

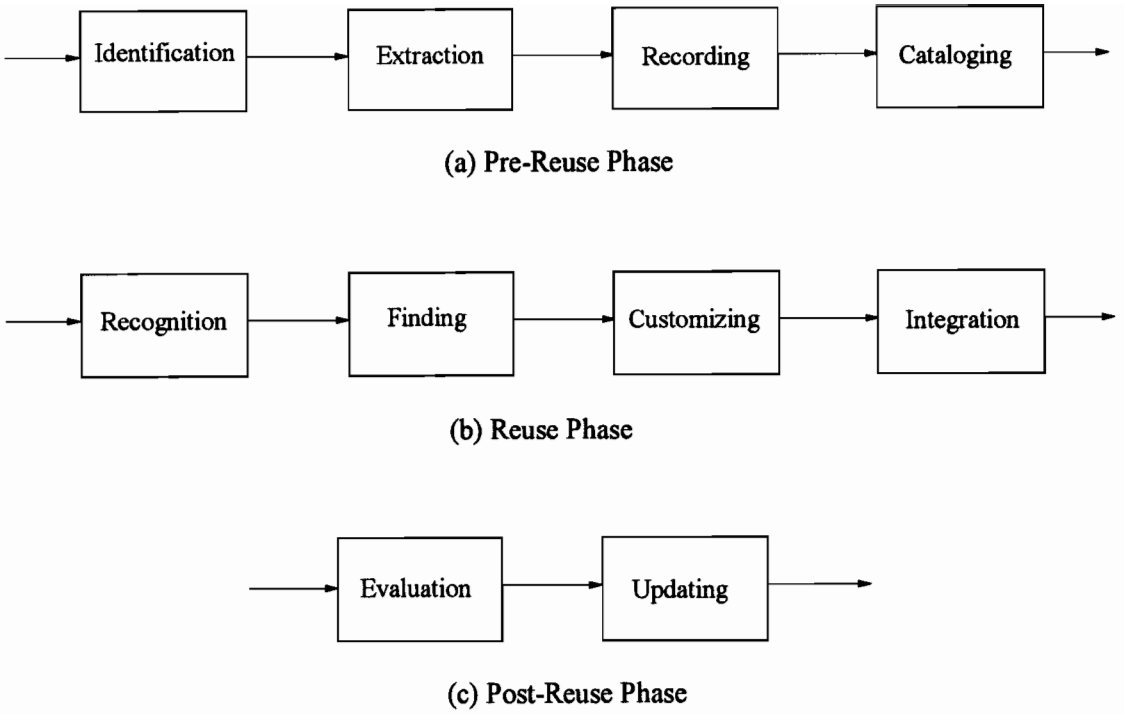

Figure 3. (a) Pre-reuse phase; (b) Reuse phase; (c) Post-reuse phase

2.2.3 Integrating Reuse Process into Development Process

The software development process and the software reuse process are highly correlated. The software reuse process can learn reusable experience from the software development process and reuse it during other software development processes. And the software development process can take advantage of the outcome of the reuse process rather than always starting from scratch, and offers its own experience as a source of the reuse process. It is possible to integrate the software development process with the reuse process to derive an incremental development process model. However, there exist some conflicts between the two processes. First, the software development process is based on perspective of single project, while the reuse process deals with more than one project, and requires a perspective that looks beyond an individual project. Second, although both

processes can be viewed as a transformation process, the software development process emphasizes the vertical transformation which refines specifications from a higher abstract level to a lower abstract level, while the reuse process emphasizes the horizontal transformation which generalizes and specializes the same abstract level of specification. Two technical strategies are adopted to obviate these two conflicts. One of them is to introduce the domain-oriented software life cycle, and add two specific stages of development: domain analysis and project recognition to support it. Domain analysis is intended to identify for producing software-related experience, while project recognition is intended to recognize for reusing existing software-related experience. The second strategy is to embed reuse processes into each phases across the whole software development process. It implies that the reuse-oriented approach can learn all levels of abstraction and reuse them at different phases.

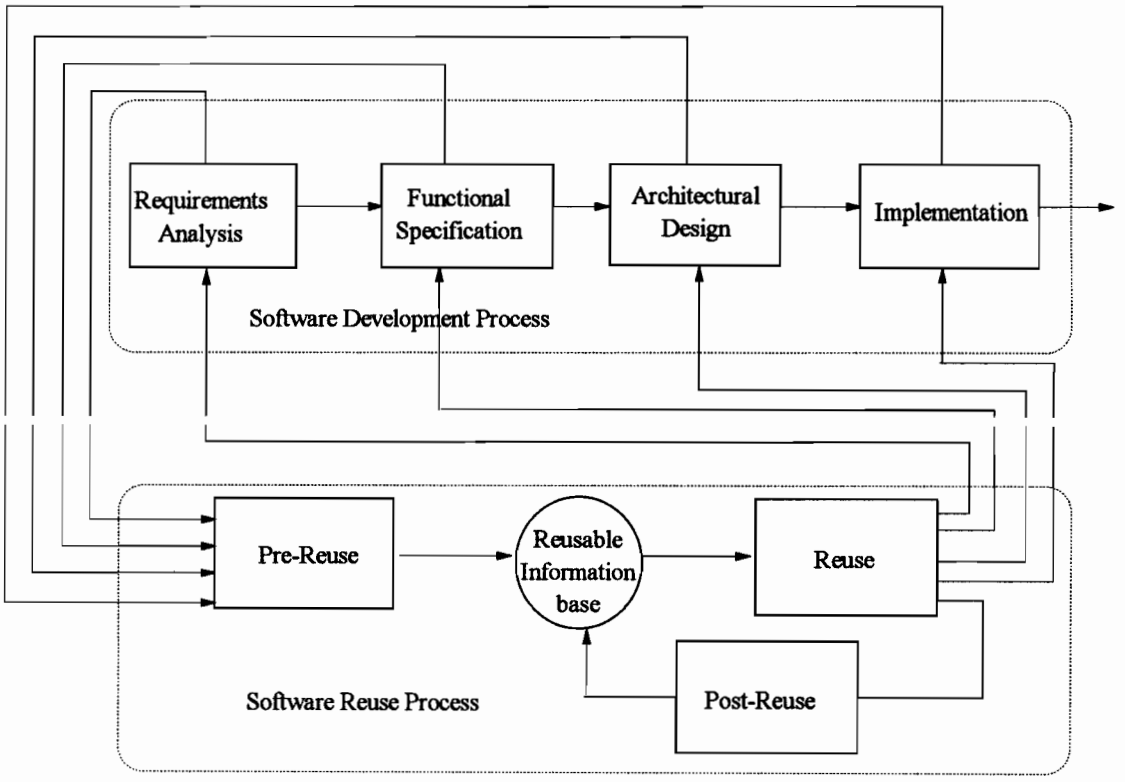

Figure 4. Integrating reuse process into development process

2.3 Technical Issues for Reuse-Oriented Approach

The reuse-oriented approach is a systematic development procedure with comprehensive reuse. It is essential to mix many technologies, process models and cultures for supporting it. The following technical issues that supports effective reuse are addressed by the reuse-oriented approach.

2.3.1 Broad-Spectrum Reuse

Reuse is a very simple concept: it means using the same thing more than once. But as far as software is concerned, it is difficult to define what is an object by itself in isolation from its context [Freeman 1983]. We have programs, designs, architecture, specifications, requirements and test cases, all related to each other. Reuse of each software object implies the concurrent reuse of the objects associated with it, with a fair amount of informal information traveling with the objects. This means we need to reuse more than code.

Traditionally, the emphasis has been on reusing code (the reusable building block) and reusing partial design (the reusable pattern). Reuse process has been limited to occurre only in the implementation phase and/or part of the architectural design phase, and ignores the importance of reuse in other phases, specially in the early phases such as the requirement analysis phase and the functional specification phase, of the software development cycle. The reuse payoff quickly reaches a ceiling that is difficult to surpass, because the reuse in earlier phases is believed to promote reuse on a large scale and at a high level.

As one of fundamental strategies, the reuse-oriented approach attempts to support the reuse process through all phases of the software development cycle. It implies that any reusable effort which is made from various phases of the software development cycle can be packaged as reusable objects, and that each phase across the software development cycle can take advantage of the most matching objects from corresponding phases. This

results in broad-spectrum reuse. We use a broad term "*software-related experience*" in a very generic sense to replace the traditional terms "building blocks" and "patterns" as reusable objects. The term "software-related experience" delineates two principal categories of reusable objects: software products and development processes. Software products include all forms of reusable sources, either concrete artifacts or documents produced during various phases of full software development cycle, or product models describing a class of concrete artifacts and documents. They bring about constructive reusability. Development processes involve all kinds of reusable information, either concrete activities of action (performed by human being or a machine) or knowledges aimed at creating some software product, or a process model describing a class of activities or knowledge with common characteristics. They lead to generative reusability.

In order to achieve the objective of broad-spectrum reuse, the key is to capture all kinds of software-related experience through the whole development cycle in appropriate reusable. Abstraction and modularity provide competent means for this task. An abstraction characterizes a class of phenomena by their common (invariant) attributes, and hides (ignores) distinguishing attributes of instances that are not common to the complete class. It allows developers to deal with different levels of macroscopic concepts for identifying various layers of reusable objects, and to understand their commonality, before going on to consider the more detailed fine-grain structures of the problem domain [Walker 1992]. Software development is considered as an iterative refinement process in which the highest abstraction (requirements specified in a problem domain) is gradually transformed into the lowest abstraction (programs to be executed on a target computer). Dijkstra described the concept of an abstract machine M(i) and program P(i) on abstract level i such that execution of P(i) on M(i) satisfies the purpose of a real program P that is to be executed on a target machine M. At the next lower level, level (i + 1), P(i + 1) can be executed on M(i + 1). If level L is the lowest level, M(L) is the target machine M, P(L) is the real program [Dijkstra 1972]. Thus we can extensively treat any valuable

information generated in a phase of the development cycle as an abstract level of reuse experience. The modularity decomposes a large and complex system into a set of small and simple self-contained mdoules. A module is a basic building block of the software system which corresponds to a single coherent abstraction. It simplifies a sophisticated system by enabling independent analysis, design and implementation of individual modules, and formularizes the construction of the system by integrating well-defined modules. In the reuse process, a module also serves as a reusable unit which encapsulates at least some valuable information for reuse. But in broad-spectrum reuse, there are many different granularities of modules associated with different levels of abstractions and various forms of modules corresponding to different reusable objects.

Reusing all kinds of software-related experience improves the reuse process in three ways. First, it extends the reuse process from a too-restrictive focus on the implementation phase to the full software life cycle incorporating the requirement analysis phase, the functional specification phase and the architectural design phase, and makes the reuse process occur in early phases so that reusable experience becomes large scale and high level. Reusing an early experience can greatly increase the likelihood of the reuse of later experiences developed from it. For example, although reusing code modules from the experience factory can certainly reduce costs, reusing the system's overall functional specification could lead to the reuse of the entire set of designs, code modules, documentation, test data, and associated user experience that was developed from that specification. The chances of cost-effective reuse are much higher, both because more experiences are reused and because the effort needed to customize and integrate those experiences into a new environment is greatly diminished. Curiously, informal reuse of early experience is actually very common, but it is often not recognized because it masquerades as code-level reuse. Informal reuse of early experience occurs primarily when highly experienced developers use their familiarity with the functionality and design of a code module set to adapt those modules to new, similar systems. Second, assembling all

kinds of experience into a special experience base actually provides existing relationships among various types of experience or among different phases. These strong relationships can assist in understanding packaged experience and identifying reusable candidates, and cut cost of tailoring selected experience and integrating experience.

2.3.2 Domain-Oriented Software Life Cycle

Reuse is the repeat use of previously acquired experience in a new situation: it consists of two subprocesses, learning and reusing, which usually don't occur in the same system development cycle. Logically, reusable experience is learnt in development of some specific system, and it is repeatedly used in development of many other similar systems. Transforming informal reuse concepts into a systematic approach requires a perspective that looks beyond the single project life cycle.

The classic software life cycle narrowly focuses on a particular project. It is not feasible to generate high-quality experience with high reusability and great reusable payoff, because it is difficult to find an appropriate reuse scope for generalizing it. It is also hard to largely reuse former efforts in developing new systems, because reuse process emerges only by accident. In order to make reuse more attractive, we need to define the applicable range of reusable objects before generating them, and to arrange later reuse locations and methods during their initial generating process. The domain-oriented life cycle seems to give the best solution.

The term "*domain*" refers to a designed collection of existing applications and anticipated opportunities for future applications with common functionality in one or more areas. Also the single project life cycle is considerably evolved to multi-project life cycle so that we refer to this multi-project evolutionary pattern as the domain-oriented software life cycle. The quality and form of reusable resources available to an individual project within a domain, and the new resources contributed as a by-product of project development alter the individual project life cycle both quantitatively and qualitatively. A

domain life cycle model formalizes typical patterns in the development of related series of application and the persistence of information from one application to the next.

The perspective of the full software life cycle reveals problems stemming from a breakdown of information traceability across individual development phases. In a similar way, viewing a series of related applications within a domain as a larger evolutionary cycle reveals problems stemming from the lack of system transition across individual system developments.

2.3.3 Multi-Organization Development Process Model

Reuse is a straightforward way of improving the software development: it can be conducted as an alternative way for development process to eliminate many duplicate and redundant works by using prior efforts. In the reuse-oriented approach, the integrated process is assumed to be based on the concept of what may be called "component engineering", in which new software system are developed by assembling "reusable components" chosen from a large, carefully designed and tested component base. It is naturally divided into distinct considerations: how to produce software components with maximum potential for reuse (development for reuse) and how to develop new systems making the most effective use of such components (development with reuse). A system development deals with two parallel goals: delivering an executable system for users and offering its new reusable resources for development of other related systems. Thus, the whole development process can be split two organizations: an experience-packaging organization and a project-generating organization. First of all, the dual organization emphasizes the reuse at beginning. Secondly, after separating these two organizations, it becomes easy to concentrate on the goals of each, and to define the best process models suitable to each organization. The experience-packaging process model consists of three stages: domain analysis, experience abstraction and experience cataloging. The project-generating process model also consists of three stages: project recognition, experience

customizing and project-integration. Finally, the separate organization simplifies the relationships between the two goals and facilitates the division of work and cooperation between phases in the domain-oriented life cycle. The primary goal of the experience-packaging organization is to package all kinds of reusable experiences as to supply them to the project-generating organization upon demand. The primary goal of the project-generating is to develop software systems by taking advantage of reusable experience provided by the experience-packaging organization. The details of each stage and their relationships will be discussed later.

2.3.4 Experience Factory

Reuse is as common as in everyday life: it is an informal sharing of software-related information among people working on the same or similar projects The informal sharing essentially needs an information base to save and manage a collection of the large amount of software-related reusable information. We borrow the term "*experience factory*" to refer to the information base in the reuse-oriented approach because we wish that the concept will lead to industrialization of software development and comprehensive reuse. The term draws from "software factory" and "component factory", and covers two aspects of meaning. "Experience" is intended to extend the basic reusable units in the information base from the traditional development end-products to domain or development knowledge, development process and other forms of reusable information. And "factory" is expected that the information base should act not only as a reusable experience repository for storing reusable experience, but also as a logical and physical experience organization for managing reusable experience.

As a experience repository, the experience factory is able to store all kinds of software-related reusable experiences in a readily available way. It implies that a experience factory plays dual roles during the development process. When developing for reuse, it gathers new reusable objects from current development; when developing with

reuse, it supplies prior reusable objects to current development on demands. Experiences manipulated by a experience factory include reusable objects from different phases across a project development and from different projects development within an application domain in varities of forms. Normally, its collection is domain-specific. We can classify a generic experience factory into three categories: general experience, domain-specific experience and project-specific experience, depending on their reuse scopes; and four levels: from analysis level to specification level to design level to implementation level, depending on their abstractions and reuse locations. The implementation level is source code, the lowest level of abstraction, and is considered the most detailed representation of a software system. In addition, complementary key information is also generated as a part of experience. Code documentation, history of design decisions, testing plans and user manuals are all essential to convey a better understanding of the whole domain.

As a logical and physical organization, the experience factory is responsible for identifying, qualifying, and classifying reusable objects for subsequent customizing and integration — by reusers — into ongoing applications development projects. It packages experiences by building informal, formal or schematized, and productized models and measures of various software process, products, and other forms of knowledge. The organization supports accumulating new experience (learning) via recording and analysis of experience, as well off-line generalizing and tailoring of experience, improving existing experience via on-line monitoring and evaluating of reusable candidates before reusing them, and retrieving the best match experience (reusing) via cataloging of experience and storing experience models in a variety of modeling notations that are tailorable, extendible, understandable, flexible and accessible.

In order to set up an actual experience factory, we should make efforts at least advancing the following basic groundwork.

First, we must determine the most suitable domain boundaries and right domain standards because the experience factory design is based on the domain-oriented life cycle.

Domain boundaries are used to control reuse scope of experience in general, and to set up a development baseline for all particular projects in advance. Domain standards are used to standardize operations on experience factory and to establish communication protocol.

Secondly, we must define a complete set of representations in order to capture all kinds of software-related reusable objects in richly machine-processible forms. The representation we are looking for must exhibit the following properties:

- the ability to represent knowledge about development process in factored form and work products from every phase of development in generalized form.
- the ability to create partial part of experience that can be incrementally extended or locally upgraded.
- the ability to allow flexible couplings between instances of experience and various interpretation they can have.
- the ability to express controlled degrees of abstraction and precision.
- the ability to represent composite structures as independent entities with well-defined computational characteristics and for these composite structures to be further composed into new computational structures with a different set of computational characteristics.

Thirdly, it is necessary to determine a classification scheme and classification rules for cataloging a large amount of reusable experience. They reflect inherent relationships among reusable individuals and imply the development context from which experience is extracted. A experience catalog will give an additional dimension for understanding reusable objects and identifying required reusable candidate without pre-training. It will also make the tracing of earlier phase or transition among projects much easier and more accurate. The main features to be sought in a classification scheme are:

expandability: It means that new classes can be added with minimal disturbance to the present collection, that is, with a minimum of reclassification problems.

adaptability: It means that the scheme can be customized to a particular environment.

consistency: It means that experience from different collections in the same class share the same attributes.

Finally, it is important to supply a series of domain-related supporting tools for facilitating or automating some operations. Tools can significantly enhance the reuse operations.

Typically, experience factory for an application domain evolves naturally over time until enough experience has been accumulated and several projects have been implemented that generic abstractions can be isolated and reused.

2.4 Reuse-Oriented Development Paradigm

Webster's say a paradigm is a pattern. From our perspective in software development, a paradigm is a pattern for a problem-solving technique. In particular, a software development paradigm specifies the steps to be followed in developing a problem into a software application. The paradigm selected determines the types of pieces that are used to present the problem and its solution, such as procedural abstractions for a procedural paradigm, entities (problem domain objects) for an object-oriented paradigm, process modules for a process-oriented paradigm, It affects the complete software development life cycle. That is, it directs the selection of analysis modeling, design methodologies, coding languages and test techniques.

A number of paradigms are in active use. They provide system developers with a large number of approaches to system decomposition. However, there is seldom an approach in which the paradigm systematically emphasizes system development for reuse and system development with reuse. We will propose a new development paradigm, reuse-oriented, in which we assumes that any system development can be resolved into a set of variable granularities of reusable modules, which we call reusable experience, and a specific set of reusable modules that can be retrieved to be integrated into a new target system. Hence it radically changes the conventional system development process. The system development

intend to successively refine the problem to the solution as well as to make the comparison of the similarities and differences between required pieces and existing reusable pieces. For the similarities we can directly reuse existing pieces . For the differences, we need to create them as new reusable experience first, then use them.

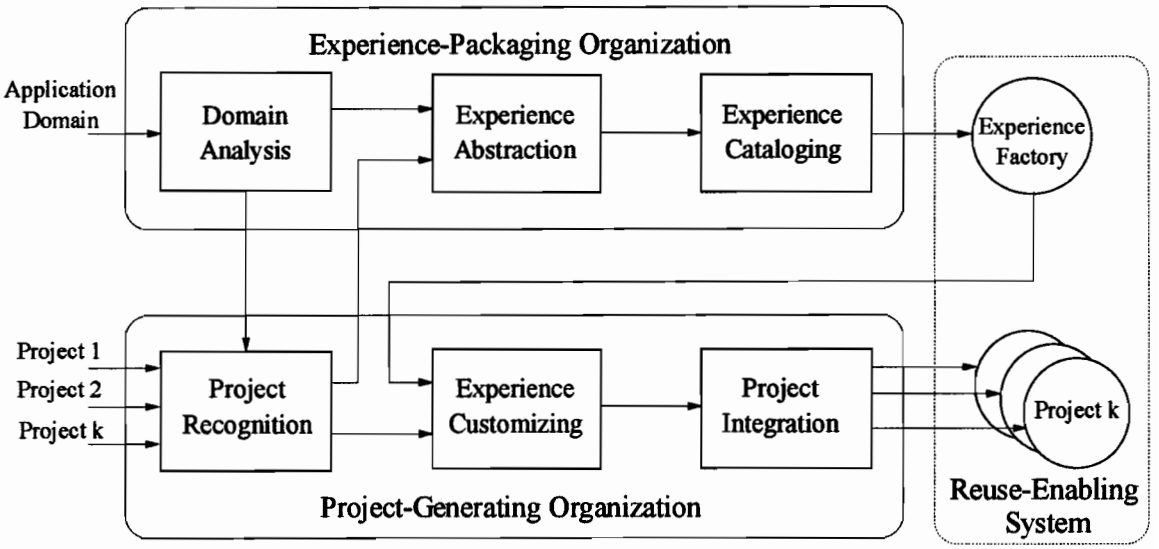

Figure 5. Reuse-oriented software development paradigm

Figure 5 shows the reuse-oriented paradigm in terms of a multi-organization framework. The reuse-based development process is separated into two interrelated organizations with best fit process models that support each organization: experience-packaging organization and project-generating organization. The experience-packaging organization specializes in producing all granularities of reusable modules from the abstract system development or a concrete project development. The project-generating organization specializes in developing a particular project by integrating the reusable modules from the experience-packaging organization.

The output of both the experience-packaging organization (experience factory) and the the project-generating organization (set of particular projects) are put together to construct a reuse-enabling system. The experience factory provides a complete reuse infrastructure for developing a new project. The set of particular projects provides examples for demonstrating how to reuse it.

2.4.1 Experience-Packaging Organization

The experience-packaging organization aims at producing new experience for current development and future reuse. It starts with a given application domain and progresses constantly with any particular project development. It includes the following three stages: domain analysis, experience abstraction, and experience cataloging.

2.4.1.1 Domain Analysis

The objective of the domain analysis stage is to identify common characteristics and similar functionalities from restricted classes of projects in an application domain with the purpose of making them reusable before developing these projects. Domain analysis plays the leading roles in packaging the high quality reusable resources with the maximum potential for reuse and the best payoff from reuse. It also facilitates the understanding, customizing and integration of packaged reusable resources during the reuse process. Domain analysis is an indispensable stage of the domain-oriented software life cycle.

In domain analysis, we try to generalize all particular projects within the domain by means of a domain model that transcends specific projects. Domain modeling is based on two aspects of domain analysis: conceptual analysis and constructive analysis, in order to capture two rather different types of information: application domain knowledge and development knowledge. Conceptual analysis focuses on identification and acquisition to specify systems in the domain. It is formualized by an explicit application domain model. Constructive analysis, on the other hand, focuses on the identification and acquisition of

the information required to implement these specifications. It is formualized by an implicit general system prototype. The domain model should be general enough to be instantiated to a broad range of target projects in project-recognition stage and expressive enough to formulate a typical solution patterns later on which can be used to produce the domain-specific reusable experience in experience abstraction stage. It should be formally represented as the top abstract level of reusable experience. After completing the creation of the underlying model, a typical solution pattern, *i.e.* a general system prototype, will be formulated in enough detail to advance the groundworks for all anticipated projects. The general system prototype actually is the abstraction of a broad range of particular projects. It exposes the essential functionalities required in the domain and processes common to projects to be provided experience abstraction stage for producing domain-specific reusable experience as a baseline of particular project development. And it hides the distinctions and particularities among projects to be left to project-generating organization for specializing them. In addition, this stage also accompanies other related activities: defining domain terminology, specifying the domain boundary, and establishing domain standards in order to advocate reusability in various fashions or at different stages of the development process.

2.4.1.2 Experience Abstraction

The objective of the experience abstraction stage is to prepare all kinds of software-related reusable experiences from different projects within an application domain, or form different phases within a particular project for reuse. A precondition for reusability in software development is the existence of reusable resources. This stage specializes in constructing reusable resources within a given domain and incrementally expanding them as new particular projects are continually developed. This stage is considered as a learning process, because the development process aims at recording all kinds of new experience for reuse instead of delivering a particular project; and its end product is a collection of

massive reusable experience rather than a executable system. There are two sources to experience abstraction stage. One is the general system prototype development from the domain analysis stage, which mainly contributes general experience and domain-specific experience. The other is particular projects development from the project recognition stage, which mainly contributes project-specific experience.

Experience abstraction is a complex combination of many activities, following a conventional software development process. First, all potentially reusable ingredients of a system development process are identified and extracted from the context of a development process or from the context of a system. Then they are generalized for application to a class of related projects. Finally the generalized reusable objects are records in formal reusable forms. Its two most important tools are abstraction and modularity, which effectively unify these activities and run through the entire stage.

Abstraction is one powerful tool that human beings possess for managing complexity and capturing generality. It arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate on these similarities and to ignore, for the time being, their differences [Hoare 1974]. In the context of software development the abstraction concerns stratification. There are principally two important forms of stratification: first, the stratification of the application domain entities into layers of complexity and compositeness; and secondly, the stratification of classes through the generalization/specialization axis [Walker 1992]. The first of these is often referred to as the aggregation, partitioning, composition or "has-a" hierarchy. It simplifies the task of understanding each development phase by partitioning it into readily assimilable chunks, by suppressing unnecessary or confusing details. This enables software development to be viewed as a iterative refinement process of abstraction. The second form of stratification is commonly referred to as the inheritance, class or "is-a-kind-of" hierarchy. It can lead generalization process which takes a solution to a specific problem and making it applicable to a class of problems.

Modularity is another tool developers possess for simplifying complexity and decomposing a large system. It already spread software engineering. Concerning the software reuse it plays a special roles because a module can serve as a basic reusable element during software development. Due to their broad spectrum of reuse, the granularity of reusable modules varies with the levels of abstraction. Each module will exhibit the following characteristics:

- Modules should. have conceptual integrity: a module is a conceptual unit in the software development process or software system which contains at least a complete object valuable for reuse.
- Modules should be highly cohesive: each module should have a central idea or purpose. The components that constitute the module should then all be related to carrying out this one central purpose. This concept is called cohesion. Cohesion refers to the degree to which the internal elements of a module are bound to or related to each other.
- Modules should be loosely coupled: modules should be as independent of each other as possible. Ideally, each module should be self-contained, and have as few references as possible to other modules. This is called coupling which refers to the degree of interconnectedness between modules [Yourdon and Constantine 1979].
- Modules should be black boxes: a black box is a system with known input and predictable output, but whose inner working are unknown or irrelevant to the users. The user's goal is to be able to perform some function with the black box, without having to understand how the box operates.
- Modules should be well documented.

In order to make each individuals higher potential reuse and less expensive modification, the following four specific features are considered during the process.

- *Project independence*: the reusable module should be as independent as possible of any particular project.

- *Complete functionality*: the reusable module should provide the complete behavior expected of instances of the concept being modeled.
- *Multiple implementations*: the reusable module should have multiple implementations that provide different run-time characteristics to allow designers a choice.
- *Generality*: A generic module specifics an algorithm without regard to a specific data type and its context.

Two additional activities are considered to facilitate producing high quality reusable experience.

- *Variants analysis*: Variants analysis is to reusable experience what requirements analysis is to traditional once-only software. Its objective is to quantify requirements for reusability up front, just as requirements analysis attempts to quantify requirements for functionality up front. Estimating reuse instances is the simplest form which consists of simply of asking questions — explicitly examining how further development or modifying efforts may be used. More elaborate forms of variants analysis require a structured format to record such information.

- *Variants specifications*: A variants specification is a requirements specification extended to include the best available information on how the activity's work products are likely to be reused. To help reusers translate these specifications into reusable experience, they are stated in terms of experience variants — functional variations of the primary experience. Experience variants can be described in many ways, ranging from explicit descriptions of multiple objects to parameterized, generic requirements.

2.4.1.3 Experience Cataloging

The objective of the experience cataloging stage is to organize effectively the large collection of reusable experiences produced during the experience abstraction stage in a readily accessible way. By making this accessible way, reusers have a leverage for ensuring reuse process cost-effective. The leverage is keeping the inherent relationships among

reusable individuals in terms of classification and cataloging. Classification displays the relationships among things and among classes of things by grouping like things together. All members of a group, or class, produced by classification share at least one characteristic that members of other classes do not. The final result is a network or structure of relationships. There are two aspects of relationships to consider: i) relationships among reusable individuals modeling a problem in the real world for capturing their natural interrelations; ii) relationships among reusable modules from different phases in a development process or from different projects in an application domain for capturing their traceability and transitions. Cataloging is locating an individual in an appropriate location within structure relationships. Sometimes, it is necessary to append more information indicating the module's external relationships. A classification scheme is a useful tool to produce systematic order based on a controlled and structured index vocabulary. It supports the archiving and retrieval of reusable experience in much the same way as the library does. Classification schemes can be either enumerative or faceted. The classification in the faceted scheme proposed by Prieto-Diaz is believed the ideal one for the reuse-oriented approach, because it is based on the assumptions that collections of reusable components are very large and growing continuously, and that there are large groups of similar components — even in very specific classes. The scheme has a component description format based on a standard vocabulary of terms, and imposes a citation order for the facet [Prieto-Diaz 1991].

2.4.2 Project-Generating Organization

Project-generating organization aims at developing a specific project by . It starts with the some particular project. Its product is a executable software system within the given domain and offer its own new experience to experience-packaging organization. It has the following three stages: project recognition, experience customizing, and project integration.

2.4.2.1 Project Recognition

The project recognition stage deals with two distinct objectives. One objective is recognizing how a new particular project can be developed by taking full advantage of reusable experiences from prior and current developments. Its output provides the later two stages of project-generating organization for reusing the pre-existing experience to current project development. Another objective is recognizing what experiences are new for current project development. Its output offers the later two stages of experience-generating organization for generating and packaging the new experiences from current project development. The reuse-oriented approach assumes that a particular projects can be constructed by integrating both prior existing reusable experience and current generating reusable experience. Project recognition plays the leading role in developing a particular project with maximal payoff of reuse, because the more effort spent in recognizing where can take reuse, what will be reused, and how it can be reused, the more likely it is to be reused, and the more costs on reusing it can be reduced. This stage starts with understanding the specific requirements of a project and instantiating the domain model generalized in the domain analysis stage to derive a specific model for the particular project. In this sense, it can be considered as a reuse process in which the reusable experience is the domain model, the top level of abstraction. The specific model will provide a means for comparison of the similarities and differences between new project and meta-system or similar projects. The recognition process results in two sorts of proceeding: direct reuse it and generate it before reuse. In order to reuse it, we need to identify the best match reusable candidate and understand it for specialization. While in order to generate it, we need to extract the reusable information and understand it for generalization.

2.4.2.2 Experience Customizing

The objective of the experience customizing stage is to bridge the gap between a given set of requirements and identified reuse candidates. Normally, the reusable experience is in general form, while This requires a precise characterization of the reuse requirements and effective mechanisms for tailoring. In addition, some refinements can be fed back to improve the quality and potential reuse of existing experience. Experience customizing is the lifeblood of reusability [Biggerstaff and Richter 1987]. It changes a static experience base to a living system of experience that spawns or evolves new experience as the requirements of the project change.

It includes four different methods: specialization, generalization, customization and enhancement. Specialization is taking a general solution and adapting it to a specific situation. It involves removing unneeded functionality, taking subset of interface data types and inputs, specializing implementation to environment and adding new properties or operations. Generalization is adopting a solution to a specific problem and making it applicable to a class of problems. It involves factoring out common characteristics, accepting more general inputs with alternative implementations selectable, and expanding to a variety of situations. Customization is creation of a specific solution from a general solution in a manner envisaged by the original design. It involves replacing abstract data types by concrete, selecting required implementation alternatives, and constraining behavior to conform to system rules. Enhancement is expanding new artifacts to the existing collections or upgrading some modules for extra functionality, better performance and tighter adherence to constraints. It is a form of elaboration [Firth 1989].

Three kinds of popular tailoring mechanisms can be applied to customizing identified experience: instantiation, modification and analogy. To an extent, the developer has anticipated instantiation by associating with the component some parameters that make it suit different contexts. Modification is an unanticipated tailoring process in which contents are changed, added, or deleted to adapt the experience to a request. Analogy is analogical

problem solving consisting of transferring knowledge from past problem solving episodes to new target problem that share significant aspects with corresponding past experience — and using the transferred knowledge to construct solutions to the target problems [Carbonell 1985]. Analogy is the most effective means of reusing the conceptual level of experience [Maiden 1991].

2.4.2.3 Project Integration

The objective of the project integration stage is to construct a particular project by integrating a set of customized reusable objects from the experience customizing stage into the context of the project development. In this stage, customized reusable objects serve as building blocks of a large system or templates of project design. This integration requires that the customized experience be embedded into the appropriate phase of the project development or assembled in the right place in the project system. After constructing the project, it continues as usual with product quality control, which includes system testing and reliability analysis, and release.

There are many alternative approaches to incremental integration. Here are three major ones:

Threads: In general, the best approach to integration is to begin by selecting a minimal set of modules that perform some central processing capability or function, called a thread. The modules selected will usually come from different levels of abstraction. Once the thread has been built in its initial form, other models can be added on to complete the thread. An advantage of this approach is that other threads from the system can be integrated in parallel and separately from the initial thread. The separately developed threads can then be integrated to construct the entire system.

Top down: The higher level of modules are integrated first, then modules from successively lower levels are added on.

Bottom up: The lower level of modules are integrated first, then modules from sucessively higher level are added on.

In practice, integration synthesizes the above three approaches. The first approach is chosen as the base of integration, and the last two approaches can complement the first approach. The best order for a particular project depends on existing reusable modules.

CHAPTER 3. CASE STUDY: RECPAM SYSTEM

3.1 Application Background

RECPAM is an acronym for Recursive Partitioning and Amalgamation. It is a tree-based modeling methodology, which provides an exploratory technique for uncovering structure in data, in statistics data analysis. RECPAM proposed by Ciampi, *et al.* is a generalization of CART, a tree-building methodology for Classification and Regression Tree (CART) [Breiman, *et al.* 1984], and its variants. In this section we will outline the basic ideals of the RECPAM methodology. The more theoretical details were fully described in [Ciampi 1991] [Ciampi *et al.* 1991, 1992].

RECPAM constructs tree models from data set of the form $\mathbf{D} = [\mathbf{U} | \mathbf{Z}]$, a matrix of measurements of the variable vectors $(\boldsymbol{u}, \boldsymbol{z})$ on a population $\mathbb{P}$ (N individuals). The **D** denotes a data matrix with rows representing observational units (OU) and columns consisting of two categories of variables **U** and **Z**. We suppose **D** to be partitioned along its columns as $[\mathbf{U} | \mathbf{Z}]$. The variables of **U**, denoted by **u** and termed *criterion variables*, are measured in order to gather information on a parameter γ, termed the 'criterion'. The variables of **Z**, denoted by **z** and termed *predictors*, are measured to contain some predictive information on γ.

The objective of a RECPAM analysis is to determine a classification of $\mathbb{P}$ into subpopulations described by statements on **z**, and homogeneous and distinct with respect to γ. This classification is constructed by a RECursive Partition algorithm which finds homogeneous subpopulations, followed by an AMalgamation algorithm which groups the homogeneous subpopulations into distinct classes. The general RECPAM method proceeds in three steps, illustrated in Figure 6. The first step grows a prediction tree by

recursively implementing binary split to locally construct a partition with maximal local information content. It results in a set of hierarchically structured binary questions, with a prediction for γ attached to any complete set of answers. Each question defines a split of an internal node of the tree, and each complete set of answers defines a terminal node or leaf of the tree. The set of the leaves of T constitutes a partition. The next two steps simplify the tree structure separately by successively cutting internal nodes of the tree and combining leaves of the tree to globally eliminate the 'negligible' information. The resulting classes are described by conjunctive and disjunctive statements involving the predictors.

Step 1: Growing of a large tree

RECPAM starts from the original population, which is identified as the root node, and searches among all admissible questions of a specified class, known as the SDQ family (family of split-defining questions). Admissibility is a local, data based restriction: a question is admissible at a node if the data at the node satisfy a certain specified condition. the question with the largest information content is selected out of the admissible SDQ's. This defines the first branching of the tree, the left branch being identified with the 'yes' answer and right one with the 'no' answer. Two children nodes are created, issuing from the two branches. The same operation is repeated recursively on the descendants, defined by subpopulations, of the root node, until nodes are reached with no admissible question: these nodes are the leaves of the large tree T_{max}. This step is terminated when all terminal nodes become leaves.

Step 2: Pruning of the large tree and selection of the honest tree

RECPAM builds a sequence of rooted subtrees (*i.e.* subtrees containing the root node) of the large tree, beginning with T_{max}, and ending with the trivial tree (*i.e.* the tree consisting of the root node only). Each subtree is obtained from the proceeding one by removing the branch with the smallest information content among those having two leaves as children. This process is known as *pruning*. The pruning sequence is determined in order of increasing information weight. It results in a sequence of nested subtrees of

increasing information loss with respect to T_{max}. Out of this sequence, the honest tree is chosen, according to a criterion that combines goodness-of-fit and economy. The *'honest tree'* implies that the simplest subtree whose information loss with respect to T_{max} is small enough to be negligible. Although such criterion should be based on cross-validation, there are two simpler, computationally cheaper alternatives: the minimum Akaike Information Criterion (AIC) and Significance Level (SL) approaches. The AIC approach consists of taking as honest tree the one such that the associated statistical model has the smallest AIC. The SL approach consists of choosing the honest tree of the pruning sequence such that its information loss respect to T_{max} is not significant at a pre-established level.

Step 3: Construction of amalgamation tree and selection of the RECPAM classification

This step is useful when the goal of data analysis is the identification of classes which are homogeneous in the same group and distinct among groups as far as the prediction is concerned. In order to obtain distinct predictions, RECPAM amalgamates the leaves of the honest tree successively, joining, at each step, the two subpopulations for which minimum information loss results in. This process is continued until the original population is reconstructed. An 'ascending' tree is thus built, similar to the trees of classical hierarchical classifications, the *amalgamation tree*. It results in a sequence of nested partitions of increasing information loss with respect to the honest tree. As in step 2, a *honest partition* is chosen from the amalgamation sequence in terms of the AIC and the SL approaches same as above. The classes of this honest partition constitute the *RECPAM classification.*

In the RECPAM user's manual, we presented a number of real examples which show how to do RECPAM analysis with the three steps [Hendricks and Lou 1993]. Here we only present a simple example to show these procedure. The real data, which is taken from Appendix VII [Breslow and Day 1987], describes nasal sinus cancer mortality in a cohort of Welsh nickel refinery workers, see Table 1. The OUs are groups of workers. The

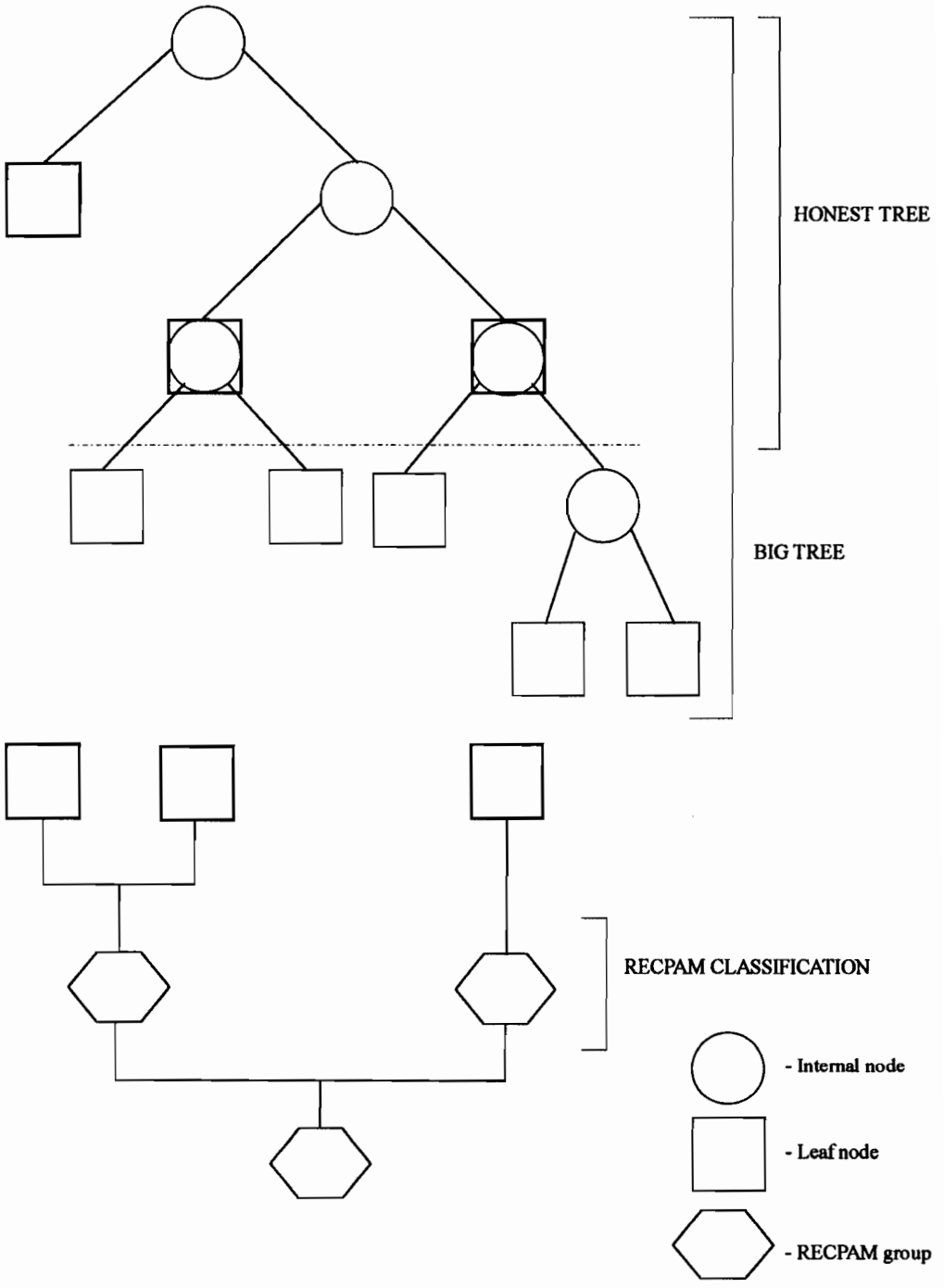

Figure 6. Illustration of RECPAM methodology

column variables include criterion variables: number of death (n) and person-years (PY), the predictor variables: age at first employment (AFE), year of first employment (YFE), exposure level (EXP), and time of first employment (TFE). Table 2 gives the categories of the variables AFE, YFE, EXP and TFE which are given in the data.

Table 1. Nasal Cancer Data Set

Nasal cancer	PY	Weight	AFE	YFE	EXP	TFE
0	1.8302	1	1	1	2	2
0	10.0	1	1	1	2	3
0	0.8739	1	1	1	2	4
0	7.1989	1	1	1	3	2
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
1	23.0416	1	4	4	3	2
0	0.219	1	4	4	3	3

The goal of the analysis is to give a predictive classification for the mortality rate (number of deaths per person-year). It is natural to assume that outcome n is Poisson with mean PYe^{γ}, where γ is the criterion, and PY as offset. In Figure 10 we give the outcome classification tree obtained from the data. It coherently organize the useful information concerning the risk of developing nasal cancel, which is contained the predictors AFE, YFE, EXP and TFE. It is obvious that the exposure level plays a crucial role, and the most left group has the highest risk, in contrast, the most right group has the lowest risk, The middle two groups share the same risks.

Table 2. Categories of the four classification variables

AFE	YFE	EXP	TFE
1 ≤ 20.0	1 = 1902 - 1909	1 = 0.0	1 = 0.0 - 19.9
2 = 20.0 - 27.4	2 = 1910 - 1914	2 = 0.5 - 4.0	2 = 20.0 - 29.9
3 = 27.5 - 34.9	3 = 1915 - 1919	3 = 4.5 - 8.0	3 = 30.0 - 39.9
4 = 35.0 - 54.4	4 = 1920 - 1924	4 = 8.5 - 12.0	4 = 40.0 - 49.9

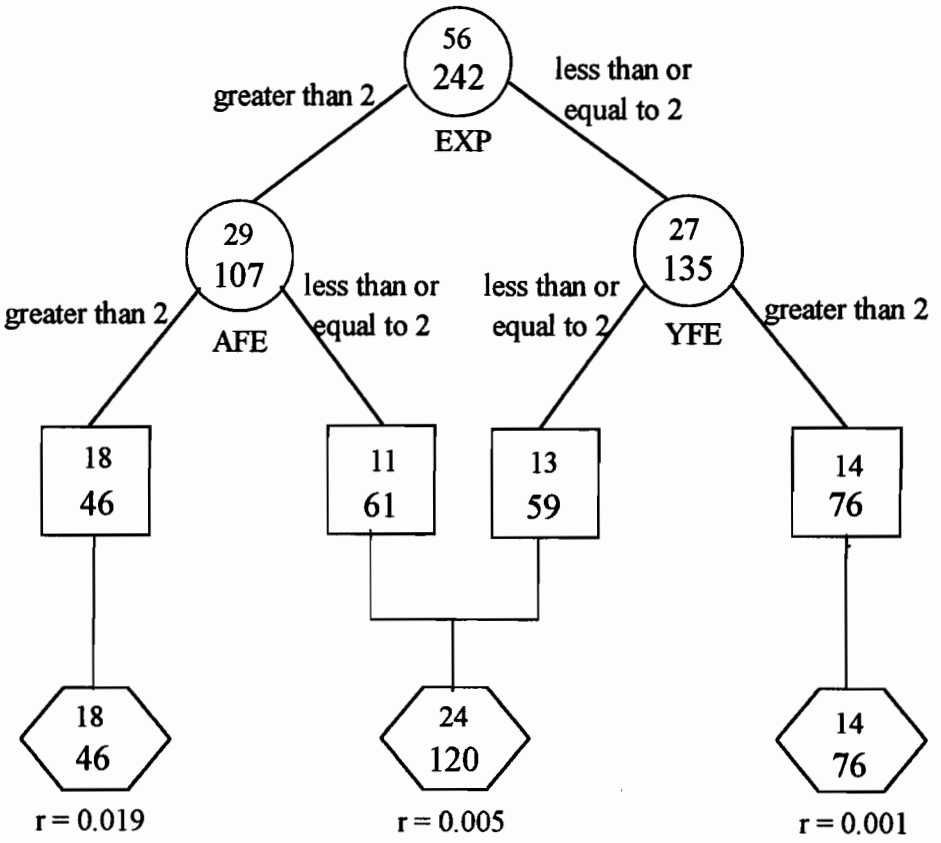

Figure 7. Outcome classification for nasal sinus cancer (outcome = death)

In Figure 7 we give the outcome classification tree obtained from the data set. The admissibility condition is specified as a minimum number of OU's (15) and a minimum number of events (10) for the leaves of the large tree. The minimum AIC rule was used for pruning and amalgamation respectively.

In order to implement above three steps, the RECPAM system also needs to supplement other three functions.

(1) Statistics model regression computation: In RECPAM, growing of a large tree needs the information content; pruning needs the information weight and AIC; and amalgamation needs the information loss and AIC. All of them originate from the same concept, information measure, which is calculated based on the statistics model regression computation. The information measure is the core of RECPAM methodology.

(2) Missing data handling: This enables the tree growing step to process the missing predictors in a data set by employing the surrogate approach of Breiman *et al* [Breiman *et al* 1984]. The approach is a strategy specific to tree construction that has much less intrinsic bias than the missing data strategies developed in the ordinary regression context.

(3) Entering tree: This is an alternative for tree growing step. A tree is constructed by recursively entering a given binary partition with prior known knowledge, instead of by recursively computing a best binary partition with maximal information content (the tree growing algorithm). It gives RECPAM methodology greater flexibility.

3.2. Objectives of RECPAM System

Owing to the increasingly important role played by computing in both theoretical and applied statistics, more and more statistics software systems have been delivered, and much more ones are expected to be quickly developed. Thus various ways to improve the productivity and quality of current statistics software development are constantly emerging. The most prevalent way is using statistics-oriented languages, such as SAS, S-PLUS, SYSTAT, to make developers to develop a software system under an high-level

integrated environment. It archives the objective in terms of statistics-specific programming capabilities hidden in the language compiler and numerous general statistical functions pre-built in the development environment. However, it still has some fatal weaknesses. First, a statistics-oriented language has adopted the reuse concept, but the reuse is bounded at low-level language primitives and code-level standard functions. There is very limited payoff from them. It doesn't support reusing at system level or subsystem level. Everything still has to be re-learnt and re-created from scratch, even though a large amount of work has been done in prior similar systems or in previous versions of the same system. Secondly, it is infeasible to produce a "open-ended" system in which system developers advance the groundworks common to the application domain and the establishment of development baseline to all designated projects, and reserve a large development space and possibility for further development. For example, a "open-ended" system allows users who are neither domain experts nor software engineers, but who are familiar with the domain and programming, to declaratively specify, implement, and modify their own applications within the domain. Thirdly, it is hard to establish a self-learning development environment in which it not only provides the initial functions, but also constantly cumulates new ones or improve existing ones. There is a belief that the reuse-oriented approach provides a best solution to these weaknesses. This belief is based on the great promises from reuse which are enumerated before.

RECPAM system is a statistics software package which implements RECPAM methodology. The following two factors motivate us to develop a reuse-enabling RECPAM system using the reuse-oriented approach propsed above.

1. RECPAM is such a general data analysis methodology that it can generate a variety of implementations which vary with statistical models, prediction models and algorithms applied to its three steps, and it can be combined with other statistics methodology, such as cross validation and bootstrap, for specific purposes. RECPAM system will deal with a large number of concrete projects which are stemmed from the general RECPAM

methodology, but give different implementation content. RECPAM system may need to be integrated with other systems. Reuse is ideally suitable for RECPAM system development, because it provides an economic and productive approach to a series of highly related projects within the same application domain, and it can remedy the defects of statistics-oriented languages. The reuse-enabling system encourages developers to constitute their own projects based on prior effort as much as possible [Lou and Ciampi 1992].

2. RECPAM is a statistics application system which roots from mathematics domain. As a domain of reuse-enabling system, RECPAM system possesses several factors which are believed to foster successful software reuse: i) the domain is relatively narrow (it contains a fairly small number of data types); ii) the domain is well understood (it has evolved over hundreds of years); iii) the underlying technology is quite static (it has reached a high level of maturity). It can be chosen as the optimum starting points to practice the reuse-oriented approach.

The reuse-enabling RECPAM system is supposed to provide two-fold functionalities:

- It functions as a regular executable software package for users to do RECPAM analysis using projects provided.
- It functions as a software reuse infrastructure for developers to develop their own new projects within RECPAM domain.

The desirable RECPAM system consists of two parts: a RECPAM project family, which collects a set of particular projects underlying RECPAM domain, and a RECPAM experience factory, which packages all kinds of software-related reusable experiences from prior development and allows new experiences from current development to continue adding in.

With the reuse-oriented approach, the development of the RECPAM system is arranged in two steps. In the first step, domain experts and software engineers cooperate to create a starter reuse-enabling RECPAM system through the development of a generalized RECPAM system that transcends any particular RECPAM application and is

called the general RECPAM system. The reuse-enabling system is well established and provides a prototype RECPAM system in terms of reusable experience. In the second step, a number of programmers who are familiar with the RECPAM domain are able to effectively and efficiently develop their own RECPAM applications on demands through the reuse of the reuse-enabling RECPAM system, in turn contribute new reusable experience in the current development to the reusable-enabling RECPAM system for future RECPAM applications. Thus the reuse-enabling system continues to grow with the development of new projects.

The RECPAM system was written in C and run under the IBM PC DOS mode. The first version of RECPAM system was compiled on Microsoft C 5.1, and the updated one was compiled on Microsoft Visual C/C++ and used the Phar-Lap's 286 DOS extender as a compiler option which can make the RECPAM system run on the DOS protected mode (up to 16 MB RAM). The source codes of RECPAM system are listed in Appendix A. The starter RECPAM system only supported a statistical model, Cox model, which presents an example for demonstration, and contained about 8K programming lines, but the present one was expanded up to 15K programming lines and added other four statistical models, Exponential model, Multi-Nomial model, Multi-Normal model and Generalized LInear Model (GLIM), as well extended local confounders into the general prediction model. From the viewpoint of users, the system is designed the menu-driven operation mode in which each menu item drives an corresponding executable program for a required function or option. RECPAM analysis uses the data-oriented form in which the statistics model and prediction model of RECPAM analysis are determined by the input data file specifications. From the viewpoint of developers, the system is designed to be reusable, domain-specific and extensible. Developers can economically add new RECPAM application projects, modify pre-existing RECPAM projects or integrate the RECPAM domain with other application domains. The starter RECPAM system, which generalized an abstract RECPAM implementation from different statistical models, various prediction

models and variant information measures, and advanced groundwork common to a designated class of RECPAM application projects, was created by a team in which members are either domain expert or software engineer. After that, the reusable system system allows a number of persons who are neither domain experts nor software engineers, but who are familiar with the RECPAM methodology and are experienced programmers, to develop concrete RECPAM application projects with their own interest. For the RECPAM system, a experienced programmer means that a person has to know how to program in C and understand the object-oriented technology and basic software engineering principles.

3.3. Development of RECPAM System

Reuse-oriented approach only provides a paradigm and several guidelines for the development of reuse-enabling system. In practice, we will also integrate an appropriate object-oriented methodology into the reuse-oriented approach to make it reality. In development of the RECPAM system, the object modeling technique proposed by Rumbaugh *et al.* [Rumbaugh 1991] is chosen as the best candidate. In this section, first of all, the object modeling technique is outlined. Then how the reuse-enabling RECPAM system is achieved by the reuse-oriented approach blending with the object modeling technique is briefly described. Finally how the development of a series of concrete RECPAM projects are improved by the reuse-oriented approach with the prior reuse-enabling RECPAM system is illustrated.

3.3.1. Object Modeling Technique

The Object Modeling Technique (OMT) is an object-oriented approach to software development based on modeling objects from the real world, then using the models to build a language-independent design organized around those objects, and gradually adding detail to transform the models into an implementation. It integrates the techniques of

object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented programming (OOP) into the appropriate phases of software development cycle. An object-oriented approach is a new way of building a system around objects rather than around functionality. Objects, which combine both data structures and behaviors in a single entity, serve two purposes: They promote understanding of the real world and provide a practical basis for computer implementation. Besides these, they also can be served as the reusable elements in reuse-oriented approach. Object orientation facilitates the implementation of reuse-oriented approach through encapsulation which accomplishes three key things: (i) it hides complexity in two ways: by concealing internal data structures and functions, and by providing a programmer interface that does not require knowledge of a object's internal workings. (ii) it discourages from unnecessarily tempering with data structures and functions that are already functional, and provides, in a roundabout way, shortcuts to manipulating data structures. (iii) the self-contained nature of encapsulated objects encourages the use and reuse of already developed modules.

The fundamental concept of OMT is the model. A model is an abstraction of something for the purpose of understanding it before building it. Because a model focuses on the essential, inherent aspects of an entity and ignoring its accidental properties and nonessential details. It is easier to manipulate than the original entity. The model has two dimensions of prospects: a prospect of a system and a prospect of development cycle.

From the prospect of a system, the OMT methodology combines three kinds of formal models to describe an application system from different views: the object model, describing the objects in the system and their relationships; the dynamic model, describing the interactions among objects in the system; and the functional model, describing the data transformation of the system.

1. *Object model*: The goal of object model is to capture those concepts from the real world that are important to an application. It describes the structure of the objects in a

system—their identity, their relationships to other objects, their attributes, and their operations.

2. *Dynamic Model*: The goal of dynamic model is to capture control that describes the sequences of operations that occur, without regard for what the operations do, what they operate on, or how they are implemented. It describes those aspects of a system concerned with time and the sequencing of operations—events that mark changes, sequences of events, states that define the context for events, and the organization of events and states.
3. *Functional Model*: The goal of functional model is to capture what a system does, without regard for how or when it is done. It describes those aspects of a system concerned with transformations of values—functions, mappings, constraints, and functional dependencies.

The three models are orthogonal parts of the description of a complete system and are cross-linked. Each model describes one aspect of the system but contains references to the other models. The object model describes data structure that the dynamic and functional models operate on. The operations in the object model correspond to events in the dynamic model and functions in the functional model. The dynamic model describes the control structure of objects of objects. it shows decisions which depend on object values and which cause actions that change object values and invoke functions. The functional model describes functions invoked by operations in the object model and actions in the dynamic model. functions operate on data values specified by the object model. The functional model also shows constraints of object values.

From the prospect of a development cycle, the OMT methodology consists of four phases: analysis, system design, object design and implementation. Three models of the system are developed initially and then gradually refined in all these phases. At each phase, they provide the different levels of abstraction for the system.

1. *Analysis*: the building of a model of real-world situation, based on a statement of the problem or user requirements.
2. *System design*: the partitioning of the target system into subsystems, based on a combination of knowledge of the problem domain and the proposed architecture of the target system.
3. *Object design*: construction of a design, based on the analysis model enriched with implementation detail, including the computer domain infrastructure classes.
4. *Implementation*: translation of the design into a particular language or hardware instantiation, with particular emphasis on traceability and retaining flexibility and extendibility.

3.3.2 Creation Step of the Reuse-Enabling RECPAM System

A complete development cycle of the RECPAM system is divided two steps: the creation step of the reuse-enabling RECPAM system and the development step of the reuse-enabling system. In the first step, the considerations of reusability in the development of the general RECPAM system are exploited with emphasis on the experience-packaging organization instead of the project-generating organization. In order to facilitate developing the general RECPAM system, a particular project, RECPAM implementation for the Cox model, was selected as an illustrative example.

Domain Analysis

Domain analysis stage is the heart of experience-packaging organization. It also provides the basis of project-generating organization. The stage starts from bounding the domain, follows by modeling the domain, and concludes with establishing the domain standards.

(1) Bounding the domain

The first substage is specifying boundaries of the domain in order to limit the type and amount of information to be treated in an application domain. These boundaries determine

reuse scopes of the domain, which control applicability and potential reusability of reusable experience generated in next stage. They highly depend upon the understanding of the domain characteristics and the requests common to anticipated applications. Normally the activity needs the trade-off between domain generality and reusability as to define the most proper domain boundaries. Here we listed three of RECPAM domain boundaries as examples. Tree structure in the RECPAM is confined to a strictly binary tree pattern in which each node has either a binary aprtition (internal node) or no partition (leaf node). There are only two types of prediction trees constructed from the data using RECPAM methodology: the classification tree that predicts a categorical response, and the regression tree that predicts a continuous response. the RECPAM tree-modeling method can be applied to two categories of data analysis: outcome classification that constructs homogeneous strata with respect to response and subgroup analysis that constructs homogeneous groups for which the effect of a given factor vary systematically from one group to the other. With new projects grow in number, RECPAM domain boundaries are able to continue being extended for striking a new optimum balance between domain applicability and reuse payoff.

(2) Modeling the domain

This substage is the central activity in domain analysis. A modeling domain is not a complete application but the encapsulation of the domain knowledge and engineering knowledge necessary to generalize the community common to all anticipated projects in the domain. In OMT, by collecting standard examples of implementations in the RECPAM methodology and performing system analysis of each, common characteristics from similar systems are generalized, the conceptual entities and associated behaviors common to all systems within the same domain are identified and formalized in objects and attributes, and a domain model that transcends specific applications is defined to described their relationships. We start out with four definitions for guidance in making the required conceptualization and formalization.

Object: An object is an abstraction of a set of real-world entities such that all of entities in the set—the instances—have the same characteristics; and all instances are subject to and conform to the same set of rules and policies.

Attribute: An attribute is an abstraction of a single characteristic possessed by all entities that were themselves abstracted as an object.

Identifier: An identifier is a set of one or more attributes whose values uniquely distinguish each instance of an object.

Relationship: A relationship is an abstraction of a systematic pattern of association that holds between real-world entities that were themselves abstracted as objects.

In order to capture the semantics of RECPAM domain, a description must be provided for all modeling entities. The description lists a set of short informative statements that describe the scope of the object class within the current domain, including any assumptions or restrictions on its membership or use, and describe its associations, attributes, and operations. A data dictionary is a repository of all textual descriptions for RECPAM domain model. In RECPAM domain, three fundamental object classes, data matrix, tree and partition, are identified first, and then more associated objects are instantiated from them. All of them can transcend a broad spectrum of statistical models and a variety of pediction models.

In order to capture the relationships between objects in RECPAM domain, three types of formal models: object model, dynamic model and functional model, must be provided for describing different aspects of relationships between objects in RECPAM domain. The object model represents the static, structural, "data" aspects of an application domain. The dynamic model represents the temporal, behavioral, "control" aspects of an application domain. The functional model represents the transformational, "function" aspects of an application domain. A typical software project incorporates all three aspects: It uses data structures (object model), it sequences operations in time (dynamic model), and it transforms values (functional model). Each model contains references to entities in other

models. The three kinds of models are depicted the generic RECPAM system modeling in graphic forms. They provide a prototype of the general RECPAM methodology implementation.

Object model is represented graphically with object diagrams whose nodes are states and whose arcs are relationships among objects. The object diagram is based around an extended form of Chen's entity relationship modeling [Chen 1976]. It provides an intuitive graphic notation for modeling objects, classes, and their relationships to one another. It is valuable for communicating with customers and documenting the structure of a system. Figure 8 shows a object diagram exhibiting the static structure of the general RECPAM system. There are three primitive objects: data matrix, tree and partition.

Dynamic model is represented graphically with state diagrams whose nodes are states and whose arcs are transitions between states caused events. The state diagram are specified using Harel's state diagram notation [Harel 1987]. They specify the flow of control, interactions, and sequencing of operations in a system that occur in response to external stimuli, and the states represent values of objects. Figure 9 shows a state diagram describing the behavior of the general RECPAM system. It describes the object life cycle in the RECPAM implementation.

Functional model is represented graphically with data flow diagrams whose nodes are processes and whose arcs are data flow. The data flow diagrams show the flow of values from external inputs, through operations and internal data stores, to external outputs. The DeMarco form of data flow diagrams is used [DeMarco 1978]. Figure 10 shows a overall data flow diagram presenting the functional derivation of values, without regard for when they are computed, in the RECPAM implementation.

The extensive guidelines for preparing them are given in [Rumbaugh *et al.* 1991].

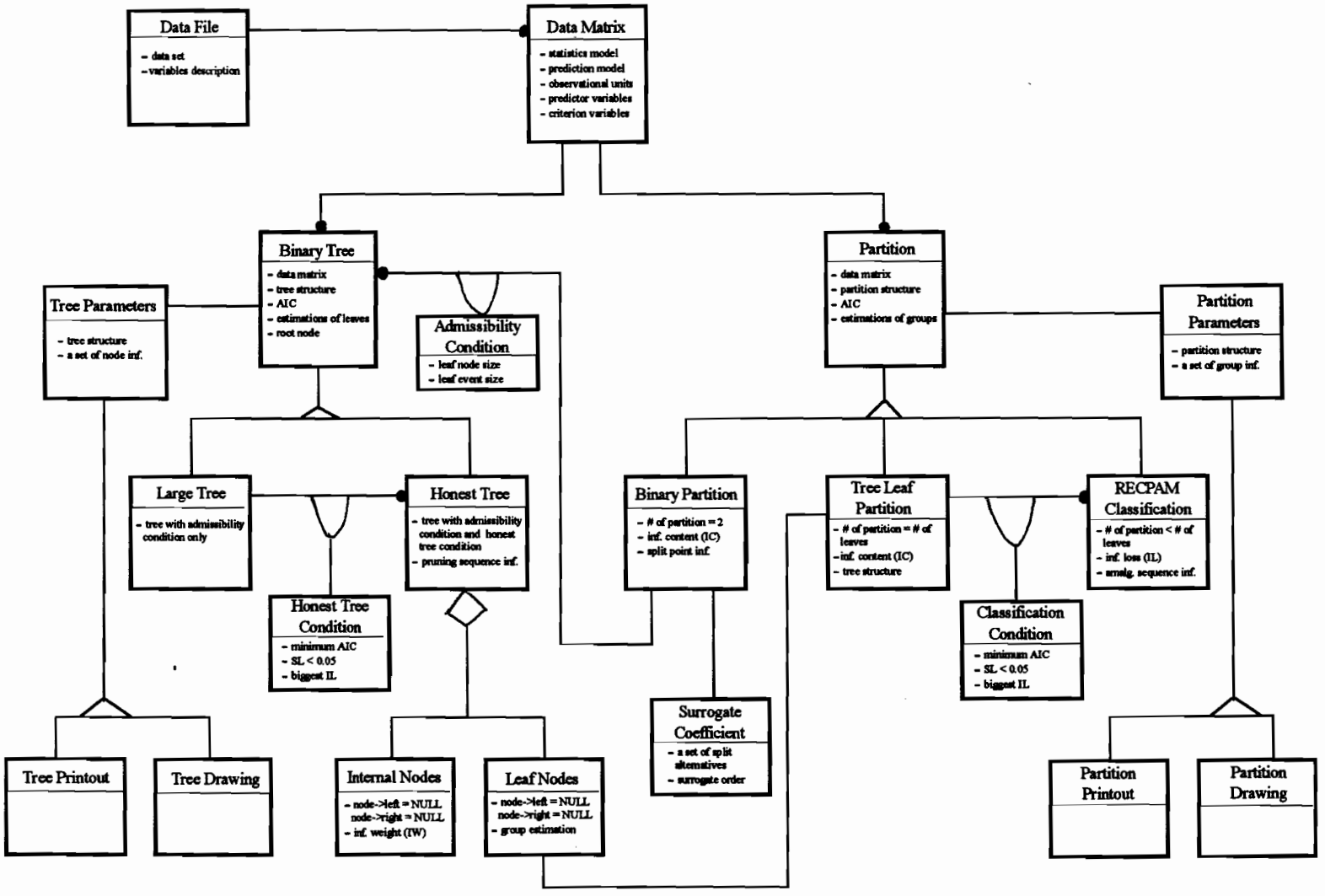

Figure 8. RECPAM object diagram for object model

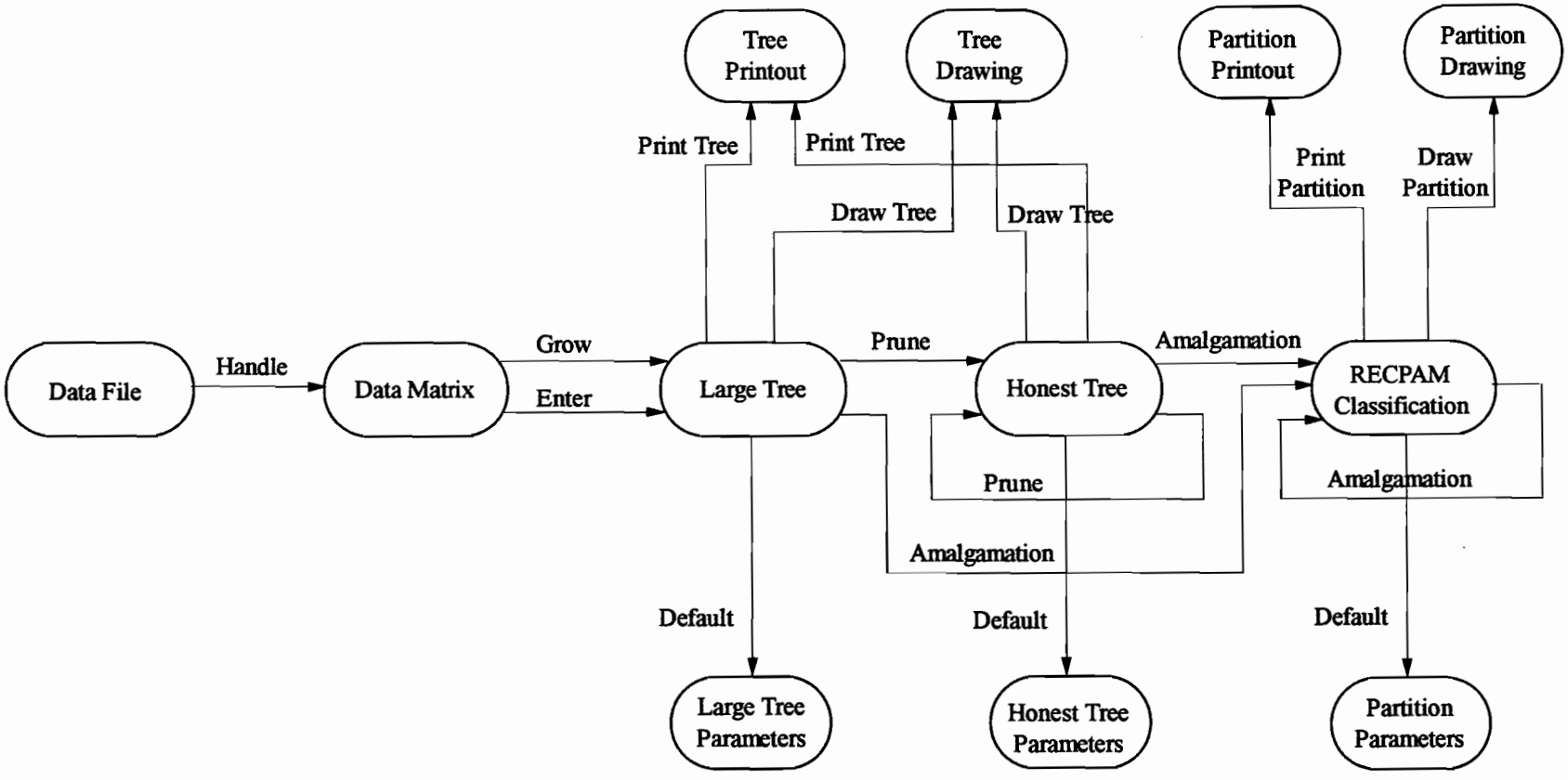

Figure 9. RECPAM state diagram for dynamic model

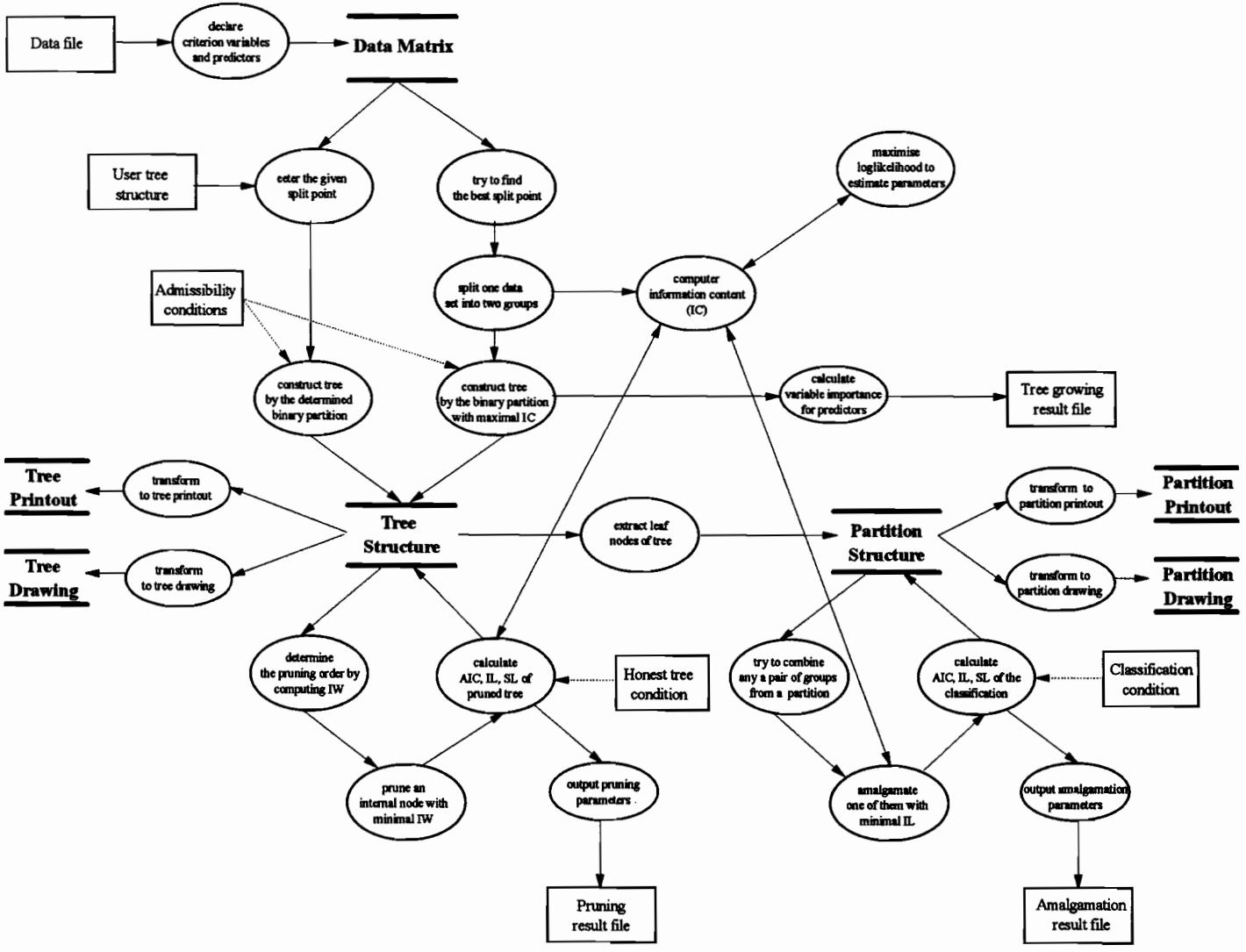

Figure 10. RECPAM data flow diagram for functional model

The data dictionary and three types of diagrams can be put together as the top abstract level of reusable experience. As the RECPAM domain model, they can be instantiated to a specific model for some particular project in the project recognition stage in project-generating organization as to recognize how to develop the project with reuse. As the general solution to RECPAM methodology implementation, they can be gradually transformed to a set of generic modules in the experience abstraction stage as to be developed in advance and set up a development baseline for a series of RECPAM related projects. As the concept-level system modeling, they can be generated a RECPAM domain specific classification scheme in the expaerice cataloging stage as to capture relationships among all kinds of software related experiences.

Based on the outcome of domain model, we attempt to formalize a typical template of RECPAM implementation, a general RECPAM system, which captures the generalities and hides the differences at different levels of abstractions. The general RECPAM system is developed before all concrete projects so that advance the basic groundwork for them by generating the domain-specific experience. It provides a development prototype of RECPAM system at implementation level instead of an executable system. In this stage, the general RECPAM system doesn't have any explicit work products, whereas it is sent to later two stages to be implicitly represented as a well-organized set of reusable experience in experience factory.

(3) Establishing the domain standards.

This substage is one of important objectives of domain analysis which enhances interconnectability and transferability of reusable individuals, as well as traceability between different phases of development or transition among different projects. The domain standards includes domain-related representations and regulations for recording all kinds of software related reusable objects, such as formats, interfaces and interconnection protocol of reusable modules, definition of domain frames and domain taxonomy for cataloging or identifying a large collection of reusable experience, and guidelines to

customize and integrate reusable candidates. They are inferred from the domain model and lay the foundations of experience factory. The most work on it coincides with generalizing domain-specific experience.

Experience Abstraction

Experience abstraction is extracting, generalizing and representing all potential reusable materials in terms of both different levels of abstractions and different granularities of modules with any system development process. In reuse-oriented approach, reusable experience stems from two types of system development: the general RECPAM system, which generates from the domain analysis stage and majors in producing domain-specific and general experiences, and a series of RECPAM related projects, which originate from the project recognition stage and major in producing project-specific and general experience.

In OMT, a development process is divided into four phases: analysis, system design, object design and implementation, which have been introduced above. Correspondingly we adopted four levels of abstraction for capturing different reusable objects from each phase across development process. Also, a development process is viewed as refinement process of system modeling, thus we can decompose the system into a number of granularities of reusable modules for satisfying different purposes of reuse.

(1) Analysis phase:

The most of works in this phase has already been done in domain analysis stage or project recognition stage. It is only left formalizing and completing problem requirements which composes of goals and constraints, in consultation with requires, users and domain experts. This is a knowledge intensive activity. Goals consists of a set of formal statements which a system must perform. They can be classified two groups: application goals and service goals. Constraints consist of a set of formal limitations which restrict the choices available to the developers. They can be classified three groups: implementation constraints, performance constraints and resource constraints.

In RECPAM system, goals and constraints both are factored into independent statements and organized into multi-level hierarchy using the formats proposed by Berzins and Luqi [Berzins and Luqi 1991]. The Table 3 shows a example which states the application goals for RECPAM meta-system. The goals and constraints identified from the meta-system should be common to all anticipated projects. A particular project can reuse them and adds more details or additional ones specific to the project. The lower level goals or constraints specify in more detail how the system is supposed to realize the higher level goals or constraints. The hierarchic structure of goals should be consistent with the customer organization and is used as reference to system organization.

Table 3. Example of Application Goals for General RECPAM System

Goal 1: Constructing prediction tree from data.

G1.1: Construction of a large tree.

G1.1.1: System can grow tree structure using recursive binary partition.

G1.1.2: System can output selected admissible questions, local information content (IC) and other related parameters for each binary partition.

G1.1.3: System can specify the admissibility conditions for stopping rule of the large tree growing.

G1.1.4: System can handle the missing predictors using surrogate variable approach of Breiman *et al* [Breiman *et al.* 1984].

G1.1.5: System can calculate the variable importance for all predictors.

G1.2: Pruning of the large tree and selection of the honest tree.

G1.2.1: System can compute the IW (Information Weight) for all internal nodes.

G1.2.2: System can prune repeatedly in order of increasing information weight (IW) of nodes from large the tree to the trivial tree.

G1.2.3: System can specify the condition of honest tree in one of AIC, SL, and User approaches.

G1.2.4: System can calculate a group of pruning parameters at each pruning step.

G1.2.4.1: Compute parameters' estimation and standard error of a tree model

G1.2.4.2: Calculate AIC (Akaike Information Criterion).

G1.2.4.3: Calculate global IC (Information Content).

G1.2.4.4: Calculate SL (Significance Level).

G1.3: Construction of the amalgamation tree and selection of the RECPAM classification.

G1.3.1: System can amalgamates the leaves of the honest tree successively by joining at each step the two subpopulations for which a minimum information loss results from the leaves of honest tree to root node.

G1.3.2: System can specify the condition of RECPAM classification in one of AIC, SL, and User approaches.

G1.3.3: System can calculate amalgamation parameters at each amalgamation step.

G1.3.3.1: Compute parameters' estimation and standard error of partition model.

G1.3.3.2: Calculate AIC (Akaike Information Criterion).

G1.3.3.3: Calculate IL (Information Loss).

G1.3.3.4: Calculate SL (Significance Level).

(2) System design phase:

This phase determines the overall organization of the system into subsystems, the external specifications of subsystems and major conceptual and policy decisions that form the prototypical framework for detailed design. It is the high-level strategy for solving the problem and building a solution. During this phase, first of all, a complex system can be successively decomposed into several simpler and smaller subsystems which encompass aspects of the system similar functionality. These subsystems are organized as a sequence of horizontal layers or vertical partitions. A subsystem is not an object nor a function but a package of classes, associations, operations, events and constraints that are interrelated (high cohesion) and that have a reasonably well-defined and small interface with other subsystems (low coupling). Each subsystem represents a sub-domain which can be developed in isolation without undue complications of having to deal with other subsystems and has its own life cycles. The conductivity of the object model can be used as the guide for the partition. The RECPAM system is broken into four principal subsystems: tree growing, tree entering, pruning and amalgamation, and four assistant subsystems: data handling, missing data handling, output handling and regression, shown

as Figure 11. Each of them can be chosen as the largest reusable end product which can be compiled independently.

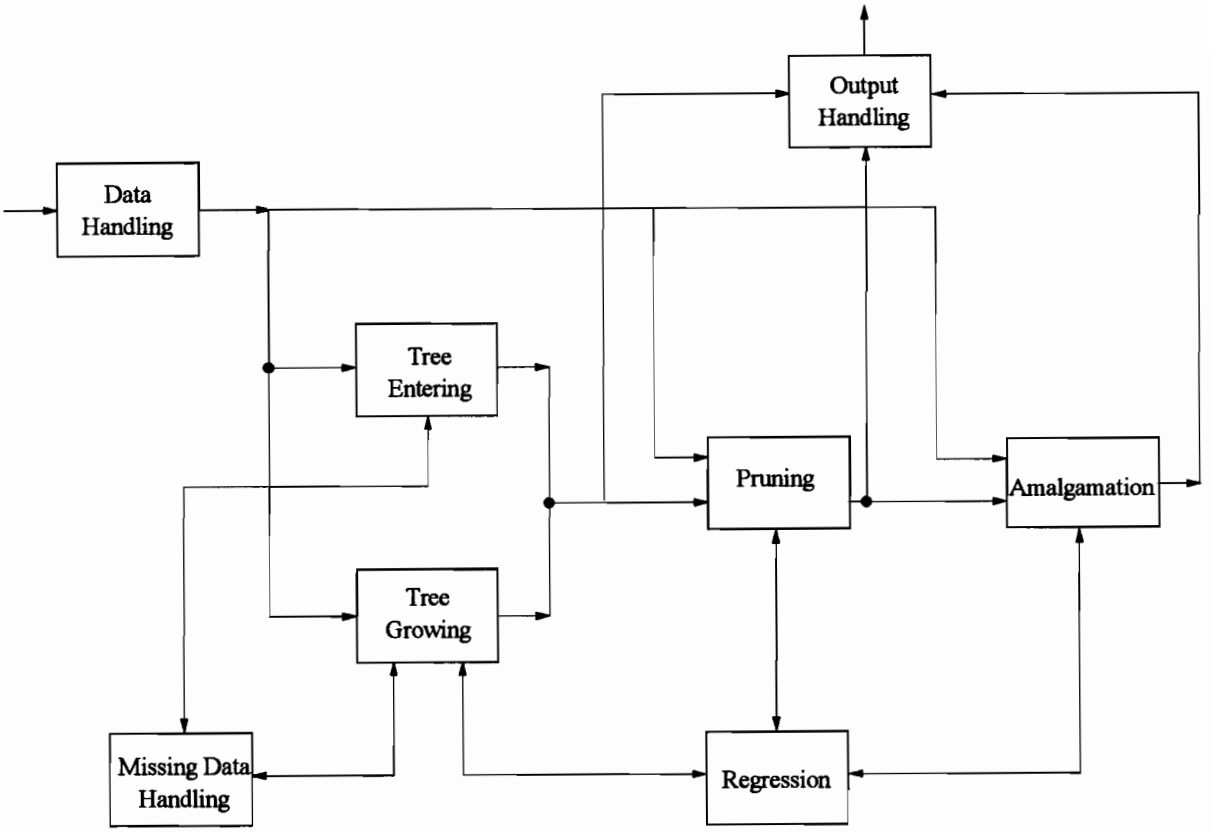

Figure 11. RECPAM system architecture

Afterwards, we separately declare each subsystem's external specifications and prototypical skeleton of detailed design using subsystem notebook. A subsystem notebook involves at least the following five kinds of information:

(i) *Service*: A service is a group of related functions that share some common purpose. It indicates the external functions of the subsystem.

(ii) *Interface*: The interface specifies the form of all interactions and the information flow across subsystem boundaries but does not specify how the subsystem is

implemented internally. It involves three types of forms: a) INPUT; b) OUTPUT; and c) CONNECT.

(iii) *Control Thread*: A thread of control is a path through a set of state diagrams on which only a single object at a time is active.

(iv) *Handling Boundary Conditions*: Three types of conditions should be considered: a) INITIALIZATION; b) TERMINATION; and c) FAILURE (Failure is the unplanned termination of a system).

(v) *Prototypical Architectural Framework*:

In RECPAM development, the subsystem notebook is simplified two items: pseudo-code algorithm and refined data flow diagram, for each subsystem, because it deals with computation objects to a great content. More details can see [Lou 1992].

(3) Object design phase:

The analysis phase determines what the implementation must do, and the system design phase determines the plan of attack. The object design phase determines the full definitions of the classes and associations used in the implementation, as well as the interfaces and algorithms of the methods used to implement operations. It adds internal objects for implementation and optimize data structures and algorithm. Object design is analogous to the preliminary phase of the traditional software development life cycle. During this phase, developers carry out the strategy chosen during system design and fleshes out the details. There is a shift in emphasis from the real-world orientation of the analysis models towards the computer orientation required for a particle implementation without descending into an individual language and a particular machine. In practice, the objects and their relationships from object model serve as the skeleton of the design. The operations identified during analysis must be expressed as algorithms, with complex operations decomposed into simpler internal operations. The classes, attributes, and associations from analysis must be implemented as specific data structures. New object classes must be introduced to store intermediate results during program execution and to

avoid the need for recomputation. After analysis we have the object, dynamic, and functional models. The object model describes the classes of objects in the system, including their attributes and the operations that they support, but it may not show some operations. We must convert the actions and activities of the dynamic model and the processes of the functional model into operations attached to classes in the object model. We use three types of primitive modules to formalize all work products at this phase: data abstraction, function abstraction, and algorithm abstraction, based on DODAN (Design Object Descriptive Attribute Notation) proposed by Yin *et al* [Yin *et al* 1988].

□ Representation of data abstraction:

The data abstraction is based on the state machine model developed by Parnas [Parnas 1972] in which the meaning of the data is expressed by the states of an abstract machine. In this approach each valid state machine is identified with a representative data object, and the expressions in the equation calculus or predicate calculus form the specifications describing how the state of data changes as a result of applying the operations.

A data abstraction consists of a data attribute and a set of operation attributes [Tanik and Chan 1991]. A data attributes has the following four kinds of attributes:

(i) *Composition attributes*:

a. DATA is the unique name by which the data abstraction will be instantiated.

b. FORMAT defines the structure of a data item, analogous to a component data type in Ada.

c. CONSTRAINT records the size, value range, and value properties, analogous to a range constraint or index constraint in Ada.

(ii) *Storage attributes*:

a. DEVICE is used to specify whether the data is stored in memory, disk, or on a tape.

b. STORAGE specifies whether the different data are stored in a linked style or a sequence.

(iii) *Operation attributes*:

a. OPS lists the operations that manipulate the data abstraction. Accessing a data item or a predicate of the data state is allowed only by the application of the operations listed in OPS.

(iv) *Similarity attributes*:

Similarity attributes denote that two data abstractions are similar with respect to certain specified criteria.

A operation attribute has the following attributes:

(i) *Invocation attributes*:

a. F-OP or V-OP is a unique name by which the data operation can be invoked. F-OP indicates that the operation changes the state of the data. V-OP indicates that the operation simply give information about the data.

(ii) *Data attributes*:

a. OPERAND specifies the objects of the data operation. The values of this attribute might not be defined in the same data abstraction.

b. RETURN specifies the results returned by the data operation. Some v-ops may produce a RETURN whose values are not defined in the data abstraction; usually such values are the predicate descriptions of data states.

(iii) *Operation attributes*:

a. PRECONDITION specifies the assumptions for executing the data operation.

b. EFFECT is only useful for f-ops. It specifies the postcondition of the data operation.

c. OPERATION specifies the specific algorithms or constraints for the data operations.

(iv) *Exception attributes:*

a. EXCEPTIONS lists the abnormal situations that may occur during the execution of the data operation and prescribes how to handle these situations.

(v) *Similarity attributes:*

Similarity attributes indicate how a previously designed data operation can be reused.

□ Representation of function abstraction:

A function abstraction is a data transformation black box whose output is determined by some abstract operations performed on its inputs. The inputs and outputs must fall within the domain and range, respectively, of the function abstraction.

A function abstraction has the following attributes [Tanik and Chan 1991]:

(i) *Invocation attributes*:

a. FUNCTION is the name by which the function abstraction will be invoked.

(ii) *Data attributes*:

a. CONSUME lists the data abstractions in the input domain of the function abstraction.

b. PRODUCE lists the data abstractions in the output range different data are stored in a linked style or a sequence.

(iii) *Operation attributes*:

a. PRECONDITION is used to specify the assumption for executing the function abstraction.

b. EFFECT is used to indicate the postcondition of the function abstraction.

c. FUNCTION is used to specify the algorithms employed to implement the data transformation.

(iv) *Exception attributes*:

a. EXCEPTIONS lists the abnormal conditions. If any of the exceptional conditions becomes true, then a specified action is to be taken.

(v) *Similarity attributes*:

Table 4 is an example a data abstraction, namely, a tree.

Table 4. Example of Data Abstraction

DATA	tree
FORMAT	tree_structure
CONSTRAINT	num_leaf ≥ 0
	num_internal ≥ 0
	num_leaf = num_internal + 1 /* strictly binary tree */
DEVICE	main_memory
STORAGE	DOUBLE_LINKED
OPS	grow_tree, read_tree, prune_tree, is_root, ...
F-OP	grow_tree
OPERAND	binary_partition, tree
RETURN	↑tree_node(root)
PRECONDITION	num_leaf = 0, num_internal = 0
	tree = {NULL}
EFFECT	tree (large tree)
	is_leaf = FALSE
OPERATION	new(tree_node)
	tree ∪ {tree_node} → tree
	if (binary_partition->split == TRUE) {
	num_internal = num_internal + 1
	tree_node->leaf = FALSE
	tree_node->information = read_split(binary_partition->information)
	tree_node->left = grow_tree(left(binary_partition), tree)
	tree_node->right = grow_tree(right(binary_partition), tree)
	}
	else {
	num_leaf = num_leaf + 1
	tree_node->leaf = TRUE
	tree_node->left = NULL
	tree_node->right = NULL
	return ↑tree_node
	}
EXCEPTIONS	if (memory_overflow == TRUE) message("overflow")
F-OP	read_tree
OPERAND	↑infile, tree
RETURN	↑tree_node(root)
PRECONDITION	num_leaf = 0, num_internal = 0
	tree = {NULL}
EFFECT	tree (large tree or pruned tree)
	is_leaf = FALSE
OPERATION	new(tree_node)
	tree ∪ {tree_node} → tree
	tree_node->information = read_node(infile)
	if (tree_node->leaf == FALSE) {
	tree_node->left = read_tree(infile, tree)
	tree_node->right = read_tree(infile, tree)
	}
	else {
	tree_node->left = NULL

```
                        tree_node->right = NULL
                        return ↑tree_node
                      }
EXCEPTIONS            if (memory_overflow == TRUE)  message("overflow")
                      if (EOF == TRUE) message("tree file error")

F-OP                  prune_tree
OPERAND               information_weight, tree_node
RETURN                Boolean
PRECONDITION          tree_node = root
OPERATION             if (tree_node->leaf == FALSE)  {
                        if (root->iw == information_weight)  {
                          tree_node->leaf = TRUE
                          tree_node->left = NULL
                          tree_node->right = NULL
                          return TRUE
                        }
                        else  {
                          prune_tree(information_weight, tree_node->left)
                          prune_tree(information_weight, tree_node->right)
                        }
                      }
                      return FALSE

V-OP                  is_leaf
OPERAND               tree_node
PRECONDITION          is_leaf = FALSE
                      tree_node = root
RETURN                Boolean
OPERATION             if (root->leaf == TRUE)
                        return TRUE
```

(4) Implementation Phase:

The implementation phase translates the object classes and relationships developed during object design into a particular programming language codes. During this phase, there are two kinds of reuse: sharing of newly-written code within a project and reuse of previously-written code on new projects. It is important to follow good software engineering practice so that implemented systems remain reusability.

In RECPAM system, the following style rules for code fragment reusability are highlighted:

- Keep method coherent. A method is coherent if it performs a single function or a group of closely related functions. If a method does two or more unrelated things, break it apart into smaller methods.
- Keep methods small. If a method is large, break it into smaller methods. By breaking a method into smaller parts, you may be able to reuse some parts even when the entire method is not reusable.
- Keep methods consistent. Similar methods should use the same names, conditions, argument order, data types, return value, and error conditions. For instance, when an operation has methods on several classes, such as computing information content in the tree growing, the pruning and the amalgamation, it is important that the methods all have the same signature — the number, types and order of arguments and type of result value.
- Separate policy and implementation. Policy methods make decisions, shuffle arguments, and gather global context. Policy methods switch control among implementation methods. Implementation methods perform specific detailed operations, without deciding whether or why to do them. Implementation methods do not access global context, make decisions, contain defaults, or switch flow of control. Do not combine policy and implementation in a single method. Isolate the core of the algorithm into a distinct, fully-specified implementation method. This requires abstracting out the particular parameters of the policy method as arguments in a call to the implementation method.
- Provide uniform coverage. If input conditions can occur in various combinations, write methods for all combinations, not just the ones that you currently need. For example, a method family corresponds with a variety of input data object. In the general RECPAM system, the tree growing, the pruning and the amalgamation can provide a class of data objects, which are from different statistical models, with variant implementation of information measure.

- Broaden the method as much as possible. Try to generalize argument types, preconditions and constraints, assumptions about how the method works, and the context in which the method operates. Often a method can be made more general with a slight increase in code. The simplest way is to includes some parameters in a method which make the method apply to a range of similar situations.
- Avoid global information. Referring to a global object imposes required context on the use of a method. Minimize external references. Often the information can be passed in as an argument. Otherwise store global information as part of the target object so that other methods can access it uniformly.
- Avoid modes. Functions that drastically change behavior depending on current context are hard to reuse. Try to replace them with models functions.
- Improve the chance of inheriting shared code. The simplest approach is to factor out the common code into a single method that is called by each method. The common method can be assigned to an ancestor class. This is effectively a subroutine call. Another approach is to factor out the differences between the methods of different classes, leaving the remainder of the code as a shared method. It is effective when the differences between method are small and the similarities are great.
- Encapsulate external code. Often you will want to reuse code that may have been developed for an application with different interfacing conventions. Rather than inserting a direct call to the external code, it is safer to encapsulate its behavior within an operation or a class.

Experience Cataloging

Experience cataloging is a indispensable stage to achieve more effective search and retrieval of reusable objects. It attempts to well organize collections of all kinds of software-related reusable objects, which are large and are growing continuously, for exposing inherent relationships among them individuals. There are two levels of

relationships to be identified in OMT: application-level relationship and implementation-level relationship.

Application-level relationship: it captures the conceptual model of application domain, such as RECPAM domain, and corresponds to interrelations of reusable objects which are from the analysis phase and the system design phase. These relationships are described in terms of the concrete problems. They provide assistance in recognizing the interactions between individual problems in the RECPAM domain.

(1) *Generalization/Specialization* allows abstractions to be defined in layers of increasing specificity. This hierarchy of abstractions provides a structuring element when we attempt to model a problem.

(2) *Aggregation* supports the development of the representation of an abstraction from several smaller and presumably simpler elements.

(3) *Classification* relates an abstraction to the instantiations of that abstraction.

(4) *Association* indicates that one abstraction serves as a holder of instances of other abstractions.

Implementation-level relationship: it captures the physical model of implementation solution, such as an executable RECPAM system, and reflects the natural position of reusable entities, which are from the object phase and the implementation phase, and their relationship to other member. The facet classification scheme is adopted to classify collections of reusable modules which are from object design phase and implementation phase of RECPAM development. In this creation step, a basic domain-oriented facet classification scheme is generated and allows to expand later as needed. Our facet scheme consists of three facets, each facet is a viewpoint toward software components:

(1) *Function* refers to the function performed. It works like a conventional library.

(2) *Object Type* refers to the template of objects to which the method belongs. In RECPAM system, it consists of three terms, each of them corresponds to a basic object of RECPAM system.

(3) *System Type* refers to subdomain which are functionally identifiable, project-independent and self-contained. In RECPAM system, it consists of eight terms, each of them corresponds to a subsystem of RECPAM system.

3.3.3 Development Step of the Reuse-Enabling RECPAM System

This step involves two parallel threads through the same development cycle of concrete projects for two highly correlated objectives. The dominant one of them is to develop new RECPAM related application products at high productivity and quality, through the project-generating organization, by taking full advantage of all forms of reusable experience packaged in the experience factory of system. It manifests great payback from the reuse-enabling RECPAM system. Another one is to contribute its own new reusable experience to the built-in experience factory for other projects, through the experience-packaging organization, as the by-products of development. It shows new investment on the reuse-enabling RECPAM system. It is obvious that the second one ensures the achievement of the first one's objective that makes a development procedure more effective and efficient. In this step, the roles of reusability in the application of RECPAM system is examined by illustrating the development of a series of actual RECPAM projects. These projects involve four different sorts of RECPAM application areas.

3.3.3.1 Bringing New Statistical Models into the RECPAM Analysis Family

The most common applications within the RECPAM domain are to make RECPAM methodology applicable to a broader spectrum of statistical models for growing a variety of classification or regression trees. RECPAM tree-modeling methodology is so general that it can be widely applied to a number of statistical models with the same tree-modeling algorithms such as the tree growing algorithm, the pruning algorithm and the amalgamation algorithm, and with the varied information measures only. Thus this sort of application just

generates statistical model specific instances of the general RECPAM methodology implementation. The reuse-oriented approach is ideally suitable for their development, because they share the greater part of the software-related experience that stems from the generality of RECPAM methodology implementation. In the reuse-oriented approach, the generalized solution to RECPAM methodology implementation has been developed in advance and has been packaged as a general RECPAM system, which can serve as the development prototype of other statistical model specific implementations, in terms of reusable forms at different abstract level during the creation step of the reuse-enabling RECPAM system. In this development step, developers can repeatedly specialize the general RECPAM system to derive many concrete implementation corresponding to particular statistical models, in a manner envisaged by the original design. This procedure involves a large amount of potential reuse opportunity, from domain model to source codes and from RECPAM expertise to development knowledge. Four statistical models: the Exponential model, the Multi-Nomial model, the Multivariate Normal model, the GLIM model are added RECPAM analysis family one by one on demand, following the initial Cox model.

In order to recognize how to specialize the general RECPAM system for a particular statistical model, development of a project should start with instantiating the RECPAM domain model. A specific model for the particular statistical model can be created as an instance of the RECPAM domain model, because the domain model was generalized to transcend all specific statistical models. Figure 12 shows the general instance hierarchy for statistical models now available. Objects and operations specific to the statistical model and their relationships are identified. It results in concept-level reuse, the highest abstract level of reuse, which can be traced to appropriate modules at lower abstract levels where instance reuse occurs and then finally be transformed to related groups of source codes, at programming-level.

Considering instanting the general RECPAM system with reuse, the development of the particular RECPAM implementation for a new statistical model in the reuse-oriented approach may actually combine three possible ways. Based on the prototype of general RECPAM implementation,

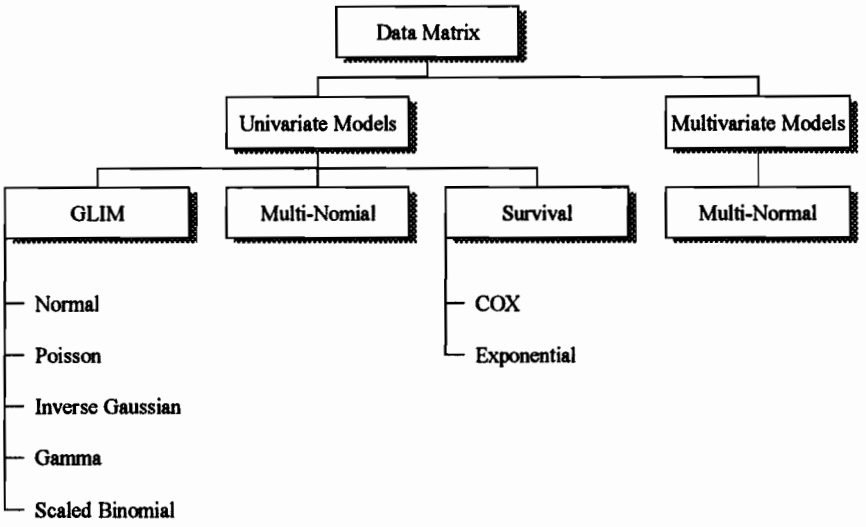

Figure 12. RECPAM statistical models instance structure

1. If the required module is not statistical model specific and dependent, we should identify it from the general RECPAM system and directly reuse it in the context of current development without any change or knowledge of its internal design. There is a large set of modules, such as the tree structure object and most methods associated it, the partition object and most methods associated it, the missing data handling subsystem, etc. which are common to all statistical models. These modules in the general RECPAM system can be transfered to the specific system. When programming in C, they can be combined in the specific system by using external functions or by including file mechanism.

2. If the required module is statistical model dependent, but it was generalized in generic forms, which unify all particular statistical models, within the general RECPAM system, reusers can specialize the generic form of module to derive its specific instance for the particular statistical model implementation. The specialization method depends strongly on the its generic form. A parameterized module is perhaps the most conventional generic form, in which each parameter provides an extra degree of freedom for increasing the module's range of pontential uses and can be selected by reusers at the time of reuse. The statistical model specific instance of the module can be configurated by declaring local parameters and messsage passing. For example, a generic class, partially describing the common data structure and parameterizing the unknowns, was defined for the data object to be analyzed. The generic class is described by a data matrix which consists of four submatrixes, respectively termed respones, confounders, determinants and predictors, and an index vector which indicates observation unit. The appearance, size and properties of these four submatrixes depend upon the particular statistical model and prediction model, and are controlled by parameters. When deriving the specific object class for an new statistical model from it, reusers can predefine parameters such as response variables, or their ranges which are statistical model specific and specialize methods assoiated with the data object, such as counting events and counting number of estimated parameters. Another generic form in common use is adopting partial specification which separates from any concrete implementation and of which implementation details can be filled in later by reusers according to their own requirements. The statistical model specific module can be completed by adding private methods for its generic specifications in terms of polymorphism and overloading mechanisms. For example, the tree growing algorthim, pruning algorithm and amalgamation algorithm all depend on the measurement of information content which is completely statistical model specific, but we have used a generalized specification to hide the different implementations in their modules. Thus they transcended a broad range of particular statistical models. When adding a new statistical

model, reusers only need to create the new statistical model specifc information content methods without repeating development of these three algorithms.

3. If the required module is statistical model specific, and it can not be generalized in generic forms, we have to generate the system specific module from more elementary modules. And this procedure can still take advantage of partial reuses. For example, regression subsystem is the basis of computating information measure, but it is completely statistical model specific. There is no module to generalize it. Thus we have to develop a particular statistical model regression subsystem for the specific system. This became the major work of each specific system development. During development of the regression subsystem, we should take the reuse of more elementary module into full account, such as Newton-Raphson method module rused in Cox model and GLIM model as a general solution to maximum likelihood estimation (MLE).

3.3.3.2 Extending the Local Confounders to the Prediction Model

Another sort of RECPAM application is to constantly update the present general RECPAM system or existing concrete projects along with the evolution of RECPAM methodology itself for amplifying its analysis capability and diversity. The reuse-oriented approach with reuse-enabling RECPAM system is believed to provide the best shortcut to their development. This belief is grounded on the fact that a required evolution can start from the basis of the general RECPAM system or closely related projects rather than always from very beginning, just like that we should stand on each other's shoulders rather than on each other's feet. Here the evolutionary reuse presents an incremental software development style which takes advantages of the inheritance relationships among requred development and pre-existing experience. The project in which the local confounders are extended to the prediction model for the measure of information content as a part of estimated parameters gave a good examples. A local confounder is a parameter assumed to be highly dependent on the predictors and should be allowed to vary "as finely as

possible" across the predictor space. In contrast, the previous confounders are referred to global confounders which don't depend on predictors and are the same for whole population. We have successfully introduced the local confounder concept into the general RECPAM system, and applied local confounders to pre-existing GLIM model and, Multi-Normal model.

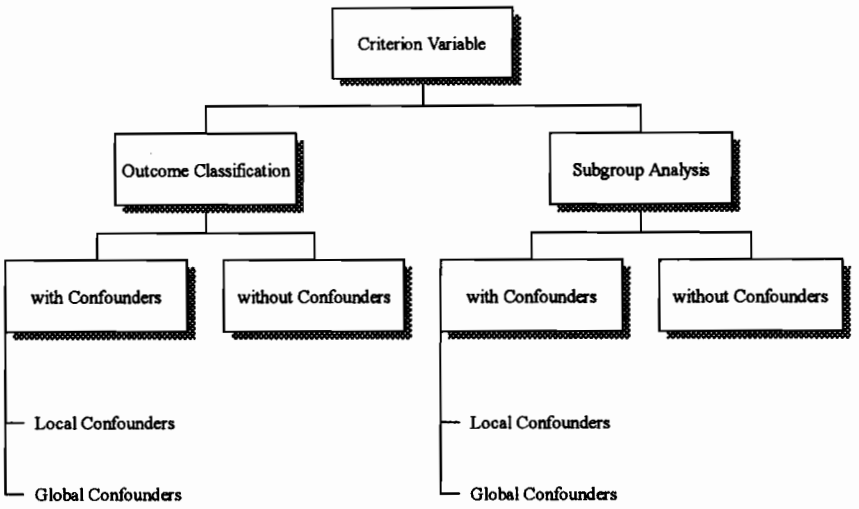

Figure 13. RECPAM prediction models instance structure with local confounders

The development of adding local confounders projects can not simply instantiate the existing generalized domain model, because the concept of local confounders go beyond its initial domain boundaries which are prescribed in domain analysis stage. But considering the reuse of previous products and knowledge, we can extend the present domain model with local confounders. A new broader domain model is evolved in terms of generalization process in which the pervious domain model is conceived of as a special instance without local confounders. Figure 13. shows the new prediction models instance hierarchy. In the new domain model, a general prediction model still involves two classes

of RECPAM analyses: outcome classification and subgroup analysis, and each of them may or may not include confounders, but the confounders can be divided into two different parts: local confounders and global confounders.

Considering evolving from the pre-existing general RECPAM system with reuse, the development of the more general RECPAM system in which the prediction model may contain local confounder(s) actually may refer to three possible ways:

1. When the required module in the new system is identical with one in the pre-existing one, we should keep the matching module of pre-existing system in the new system to inherit experience from the pre-existing system. For example, the most parts of pruning subsystem, except the method which computes the information content, can be dirctly reused in the new system without repeating previous works.

2. When the required module corresponds to one in pre-existing system, but is more general than the corresponding one, we can override it with the more general module. It implies that user can partially inherit software-related experience. For example, we extended the prior generic class of data object by separting the previous confounder matrix into two submatrixes for local confounders and global confounders. A more general class of data object was created in the same way as before. In turn, we rewrited all methods which are related to internal structure of the generic class.

3. When the required module is completely new with respect to pre-existing system, we have to develop it and then add it into new system. It can be conceived of as a unbounded generalization. For example, we added two local confounders concerned methods into pruning step. One is to obtain a finest partition from the reference tree. Another one is to transfer the finest partition to a indicator vector for computing the information content.

3.3.3.3 Adding A User-Defined Alternative of Pruning Procedure

As one of tree-modeling approaches, RECPAM is often demanded to customize its performance content with user's special concerns. For instance, user attempt to define

their own algorithms as alternatives of the original algorithm, to derive variants of current methods or to introduce some new techniques on the RECPAM methodology philosophy. The reuse-enabling RECPAM system offers a experienmental ground which encourages the practice of customizing RECPAM's performance. The reuse-oriented approach makes this sort of applications in economic fashion.

How to develop is discussed by a practical project which adds a user-defined pruning algorithm as an option in pruning step. In the original pruning algorithm, pruning sequence is obtained by the sorted list of information weights, which are globally calculated from the difference of information content at a tree with cutting the internal node with respect to large tree, for all internal nodes. While in the user-defined pruning algorithm, the pruning order is determined by globally comparing the local information contents, which are calculated at each split point during the tree growing, among all internal nodes eligible for being pruned. The rest pruning operations keep the same as original ones. Obviously, there are a large partition of overlap in the new project. The appropriate reuse will significantly improve its development. In the project recognition stage, an intersection process is first applied to realize its similarities and difference . It results in the following arrangements:

1. According to the natural feature of the project, all likely changes can be constrained only on the pruning subsystem. It means that whole project development is shrunk to a subsystem modification without knowledge of implementation of other subsystems and without affecting the behavior of other subsystems.

2. In order to make new pruning algorithm as alternative without bothering previous subsystem, we created another parallel pruning subsystem by copying the previous one. The previous pruning subsystem was used to form the basis of the new pruning subsystem which can inherit privious products for similarities, override certain methods for difference, and add any new behaviors that are required. It can be modified with far less effort than developing from scratch.

3. we can make likely changes in pruning subsystem without involving the interface modification, It can directly linked to other subsystem without recompiling other subsystem, and it make the new pruning automatically to all existing statistics model without any extra work.

3.3.3.4 Developing A Cross-Validation System on the Basis of RECPAM System

The reuse-enabling RECPAM system essentially is an elementary "open-end" system which not only supports the development of different sorts of RECPAM applications within RECPAM domain but also facilitates the creation of new systems of which domain is not constrained in RECPAM domain, but is associated with it. Sometimes, RECPAM domain is asked to be integrated with other domain or be embedded into another larger domain for study purposes. For example, in order to correct the overfit bias inherent in data-driven modeling analysis, such as RECPAM tree-modeling analysis, a cross-validation system is requested to be developed for RECPAM methodology. In the particular cross-validation system, the RECPAM methodology can be conceived of as a subdomain in the cross-validation domain in view of intentionally reusing the readily available RECPAM system in its development procedure. We have completed the development of the cross-validation system for Cox model on the basis of RECPAM system. The new system is totally separated from the RECPAM system. It is also written in C, but run under UNIX system Workstation, considering the memory limitation of DOS system. Another similar system, bootstrap system, is under the way.

Since RECPAM methodology is the major ingredient of concerned cross-validation system and acts as operating objects of cross-validation processing, reusing the readily available RECPAM system gained great benefits recognized from three aspects. First of all, it significantly reduced the amount of programming work needed on the cross-validation system. The redundant work for developing RECPAM methodology and the ground work common to both systems is eliminated by directly taking final source codes

of required subsystems or functions from RECPAM system and adapt them to the new system and new work platform. For example, total source code of cross-validation system has around 5,000 lines. More than 75% of it is based on reusing the RECPAM system. And about 50% of it is directly duplicated from the source code of RECPAM system without any modification. Five subsystems of RECPAM system are included as demand. Secondly, it enhances the development level of cross-validation system by encapsulating all functions related to RECPAM methodology as a entirety. The encapsulated entirety can separate the RECPAM subdomain from cross-validation domain, and can constitute a RECPAM subsystem of cross-validation system. The RECPAM domain and subsystem can use the readily available RECPAM system. Thus we only need to understand the responsibility of the subsystem (interface) without understanding the subsystem's internal knowledge and design. And thirdly, it ensures the inheritance of RECPAM domain knowledge and development knowledge. For example, the missing data handling algorithm as well its implementation method can be automatically brought to the cross-validation system from the RECPAM system without knowledge of its implementation as a consequence of reusing the RECPAM system. Another good example is the structure of the data object for RECPAM analysis. With reusing the RECPAM system, the generic class definition in RECPAM is introduced to the cross-validation system. The data object for cross-validation is defined as the same structure as the RECPAM system and an additional index vector for group indicator. Besides its original functions, this makes the whole data handling subsystem of RECPAM system be reused in cross-validation.

CHAPTER 4. CONCLUSIONS AND FUTURE WORK

4.1 Conclusions

The great benefits from reuse motivate us to adopt comprehensive reuse as the major means to improve present software development methodologies. The reuse-oriented approach is intended to present a general incremental development paradigm with systematic reuse. It is derived on the basis of object-oriented methodology and incorporates several outstanding technical issues. As a case study, the development of RECPAM system has demonstrated the reuse-oriented approach. The well-established starter reuse-enabling RECPAM system is created first by both domain experts and software engineers for providing a baseline to development of all concrete projects within the RECPAM domain. Then as it is repeatedly applied to the basis of development of a series of RECPAM applications by programmers who are just familiar with RECPAM domain, the reuse-enabling RECPAM system continuous to grow. We can summarize the following points:

1. The RECPAM system development confirms the rich harvest to be reaped from the reuse-oriented approach. In the short term, the reuse-oriented approach is generally more expensive than conventional approaches, because of the extra effort of producing reusable resources and managing them. But it has a long-term economic gain, especially when there are numerous anticipated projects within a mature application domain, or when applications must be continuously upgraded. For example, we attempt to implement a general statistical methodology. As the target projects increase in number, the payoff of the reuse-oriented approach will be greater. The reuse-oriented approach would be cost-effective for those application domains where a large

number of similar projects are manufactured repeatedly, such as the RECPAM domain.

2. The reuse-oriented approach introduces a new form of software product, the reuse-enabling system, which can provide users not only an executable software system, but also a domain-specific integrated development environment. A reuse-enabling system is best suitable for neither software engineers nor domain experts, but experienced programmers who are familiar with the domain, to develop their own projects on the pre-established development baseline. It ensures the incremental development and the parallel development in the form of reusable experience.
3. Making reuse attractive in the reuse-oriented approach is largely an intellectual activity of finding the right technical culture, the appropriate domain boundaries and domain standards, the right representation of reusable experience and other frastructure. It is not simply a matter of following the development paradigm with the several supporting guidlines. The details of sucess are defined by comprehensive technical analysis and a strong focus on the application domain.
4. Some of the key factors that foster successful reuse are realized:
    - Narrow domain: Narrow domain allows the use of reusable objects on larger scale, at high abstract level or in earlier phase, and increase the amount of the target application that can be constructed from reused objects. The appropriate domain boundaries dramatically increase the reuse ratio in development of new projects and decrease the reuse cost of both development for reuse and development with reuse. A successful application domain is narrower than expected. It is necessary to narrow it down to a specific product family, rather than to cover a broader field.
    - Well understood domain: Without a good model of the application domain it is difficult to derive appropriate, widely applicable and high quality reusable experience. Without a good understanding of domain, it is infeasible to take full advantage of reuse to improve the development.

- Slowly changing domain technology: Reusable experience decays over time and rapid changes in underlying domain technology force a reuse-enabling system to decay and thereby, depreciate in value too rapidly to recapture, in savings, the cost to construct the system in the first place.
- Well established experience factory: The more efforts we make on it, the much more payoff they will return. The experience factory is a bridge between two organizations of development in the reuse-oriented approach. It is the guarantee of producing high-quality reusable experience. It also is pre-condition of making reuse attractive.
- Economies of scale: Build reuse-enabling systems to service areas where there is lots of opportunity to reuse the experience. It would deserve application if three or more manufacturing cycles are expected for the product family.

5. Object-oriented methodology is essential. Its value lies in providing the conceptual foundation for reuse-oriented approach and facilitating organization, representation and operation of reuse. A practicable reuse-oriented approach is established on some particular object-oriented methodology.

4.2 Future Work

Although the reuse-oriented approach has been formalized and practiced in real software system development, many areas of this approach remain to be fully developed. This is merely a good starting point. More research and practice works are expected to be exerted.

First, we should ground on the current RECPAM reuse-enabling system to develop more concrete projects within RECPAM domain for exploiting its potential reusability and applicability. For the moment, there are two new sorts of application projects: (i) the data object to be is not coned data matrix formats. for example, point process statistics model has recurrent events for each observation units [Ciampi *et al.* 1992]; (ii) RECPAM system

works as a subdomain in a large system development. A project, bootstrap for RECPAM tree-modelling approach, is under the way.

Second, we will continue practicing the reuse-oriented approach in more application domain or spreading the reuse-enabling system for more developers to reuse as to improve reuse-enabling system and reuse-oriented approach.

Finally, this paper focused on an overall development paradigm of reuse-oriented approach and a set of guidelines of reuse operations, and identified related activities for each stage in the framework. But no methodology or any kind of formalization for each stage is yet available. This case study concentrated on the outcome, not on the process. We have to formalize it to provide a complete reuse-oriented approach.

REFERENCE

Arango, G., Baxter, I., Freeman, P., and Pidgeon, C., "TMM: Software Maintenance by Transformation", IEEE Software, Vol. 3, No. 3, pp. 27-39, May 1986.

Arango, G., "Domain Engineering for Software Reuse", Ph. D. Thesis, Department of Imformation and Computer Science, University of California, Irvine, 1988.

Arango, G., "Domain Analysis-From Art Form to Engineering Discipline", 5th International Workshop on Software Specification and Design, ACM SIGSOFT Software Engineering Notes, Vol. 14, No. 3, pp. 152-159, May 1989.

Basili, V.R. and Rombach, H.D., "Support for Comprehensive Reuse", Software Engineering Journal, Vol. 6, No. 5, pp. 303-316, Sep. 1991.

Basili, V.R., Caldiera, G. and Cantone, G., "A Reference Architecture for the Component Factory", Technical Report CS-TR-2607, Department of Computer Science, University of Maryland, College Park, Maryland, Mar. 1991.

Basili, V.R. and Rombach, H.D., "Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment", Technical Report CS-TR-2158, Department of Computer Science, University of Maryland, College Park, Maryland, Dec. 1988.

Bauer, F.L., "From Specifications to Machine Code: Program Construction through Formal Reasoning", Proceedings, Sixth International Conference on Software Engineering, pp. 84-91, 1982.

Bemer, R.W., "Position Papers for Panel Discussion: The Economics of Program Production", in Information Procession 68, North-Holland, Amsterdam, pp. 1626-1627, 1969.

Berzins, V. and Luqi, "Software Engineering with Abstractions", Addision-Wesley Publishing Company, 1991.

Biggerstaff, T.J. and Perlis, A.J., "Foward: Special Issue on Software Reusability". IEEE Transactions on Software Engineering, Vol. 10, No. 5, pp. 474-476, Sep. 1984.

Biggerstaff, T.J. and Richter, C., "Reusability Framework, Assessment, and Directions", IEEE Software, Vol. 4, pp. 41 - 49, Mar. 1987.

Boehm, B.W., "Software Engineering", IEEE Transactions on Computers, Vol. C-25, No. 12, pp. 1226-1241, Dec. 1976.

Boehm, B.W., "A Spiral Model of Software Development and Enhancement", ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 4, pp. 14-24, Aug. 1986.

Boldyreff, C., "Reuse, Software Concepts, Descriptive Methods and the Practitioner Project", ACM SIGSOFT Software Engineering Notes, Vol. 14, No. 12, pp. 25-31, 1989.

Boyle, J.M. and Muralidharan, M.N., "Program Reusability through Program Transformation", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, pp. 574-588, May 1984.

Breslow, N.E. and Day, N.E., Statistical Methods in Cancer Research, Heseltine, E. (ed.), IARC Scientific Publications, Vol. 2, No. 82, Oxford University Press, 1987.

Breiman, L., Friedman, J.H., Olshen, R.A. and Stone, C.J., "Classification and Regression Trees", Waldworth International Group, Belmont, California, 1984.

Burton, B.A., Aragon, R.W., Bailey, S.A., Koehler, K.D., and Mayes, L.A., "The Reusable Software Library", IEEE Software, Vol. 4, No. 4, pp.25-33, Jul. 1987.

CAMP, Common Ada Missile Packages, Final Tecnical Report, Vols. 1, 2 and 3. AD-B-102 654, 655, 656. Air Force Armament Laboratory, AFATL/FXG, Elgin AFB, FL, 1987.

Carbonell, J.G., "Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition", Tecnical Report CMU-CS-85-115, Computer Science Department, Carnegie-Mellon University, Mar. 1985.

Champeaux, D.D. and Faure, "A Comparative Study of Object-Oriented Analysis Methods", Journal of Objected-Oriented Programming, Vol. 5, No.1, pp. 21-33, 1992.

Cheatham, T.E., Jr., "Reusability Through Program Transformations", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, pp. 589-594, Sep. 1984.

Chen, P.P., "The entity-relationship model — toward a unified view of data", ACM Transaction, Database System, Vol. 1, No. 1, pp. 9-36, 1976.

Ciampi, A., Lou, Z., Lin, Q. and Negassa, A., "Recursive Partition and Amalgamation with the Exponential Family: Theory and Applications", Applied Stochastic Models and Data Analysis, Vol. 7, No. 2, pp. 121-137, 1991.

Ciampi, A., "Generalized Regression Trees", Computational Statistics and Data Analysis, Vol. 12, No. 1, PP. 57-78, 1991.

Ciampi, A., Hendricks, L., and Lou, Z., "Tree-Growing for the Multivariate Model: The RECPAM Approach", Dodge, Y. and Whittaker, J. (*eds.*), in Computational Statistics, COMPSTAT, Physica-Verlag, A Springer-Verlag Company, Vol. 1, pp. 131-136, Aug. 1992.

Ciampi, A., Negassa, A. and Lou, Z., "Tree-Structured Prediction for Censored Survival Data and Cox Model", 1993. (submitted)

Ciampi, A., Hendricks, L. and Lou, Z., "Discriminant Analysis for Mixed Variables: Integrating Trees and Regression Models", Cuadras, C.M. and Rao, C.R. (*eds*), in Multivariate Analysis: Future Directions 2, Elsevier Science Publishers B.V., 1993.

Ciampi, A., Dougherty, Lou, Z., Negassa, A. and Grondin, J., "NHPPREG: a Computer Program for the Analysis of Nonhomogeneous Poisson Process Data with Covariates", Computer Methods and Programs in Biomedicine, Vol. 38, pp. 37-48, 1992.

Cusumano, M.A., "The Software Factory: A Historical Interpretation", IEEE Software, pp. 23-30, Mar.1989.

DeMarco, T., "Structured Analysis and System Specification", Yourdon Press, New York, 1978.

Dijkstra, E.W. Notes on Structured programming. In Structured programming. New York: Academic, 1972.

Dubinsky, E., Freudenberger, S., Schonberg, E. and Schwartz, J.T., "Reusability of Design for Large Software Systems: An Experiment with the SETL Optimizer", in Software Reusability: Volumn I, Concepts and Models, Biggerstaff, T. J. and Perlis, A. J. (eds.), ACM Press, Addison-Wesley Publishing, pp. 275-293, 1989.

Firth, R., Software Design, Lecture Notes, Software Engineering Institute, Carnegie Mellon University, 1989.

Frakes, W.B., Biggerstaff, T.J., Prieto-Diaz, R., Matsumura, K. and Schaefer, W., "Software Reuse: Is It Delivering?", IEEE 13th Internation Conference on Software Engineering, pp. 52-59, 1991.

Freeman, P., "Reusable Software Engineering: Concepts and Research Directions. In Proceeding of ITT Workshop on Reusability in Programming. ITT, Startford, Conn., pp.129-137, 1983.

Gladden, G.R., "Stop the Life Cycle, I want to Get Off", ACM Software Engineering Notes, Vol. 7, No. 2, pp. 35-39, 1982.

Gomaa, H., "A reuse-oriented approach for structuring and configuring distributed applications", Software Engineering Journal, Vol. 8, No. 2, pp. 61-71, Mar. 1993.

Harel, D., "Statecarts: a visual formalism for complex systems", Science Computer Program, Vol. 8, pp.231-274, 1987.

Hendricks, L. and Lou, Z., RECPAM User Manual, Montreal Children's Hospital Research Institute, Montreal, 1993.

Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", Comm. ACM, Oct. 1974.

Horowitz, E. and Munson, J.B., "An Expansive View of Reusable Software", IEEE Transaction on Software Engineering, Vol. SE-10, No. 5, pp. 477-487, Sep. 1984.

Kant, E. and Barstow, D.R., "The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Programming Synthesis", IEEE Transactions on Software Engineering, Vol. SE-7, No. 5, pp. 458-471, Sep. 1981.

Kim, K.H., "A Look at Japan's Development of Software Engineering Technology", Computer, Vol. 16, No. 5, pp. 26-37, May 1983.

Korson, T. and McGregor, J.D., "Technical Criteria for the Specification and Evaluation of Object-oriented Libraries", Software Engineering Journal, Vol. 7, No. 2, pp. 85-94, Mar. 1992.

Lanergan, R.G. and Poynton, B.A., "Reusable Code: The Application Development Technology oh the Future", In Proceedings of the IBM SHARE/GUID Software Symposium, Monterey, Calif.:IBM, Oct. 1979.

Lenz, M., Schmid, H., and Wolf, P., "Software Reuse Through Building Blocks: Concepts and Experience", IEEE Software, Vol. 4, No. 4, pp. 34-42, Jul. 1987.

Lou, Z., Appendices in "Contructing Prediction Trees from Data: RECPAM Approach", Ciampi, A., in Computational Aspects of Model Choice, Physica-Verlag, A Springer-Verlag Company, Vol. 1, pp. 144-150, 1992.

Lou, Z. and Ciampi, A., "Reuse-Oriented Approch in Developing Statistics Software", in Proceedings of the 24th Symposium on the Interface, Computing Science and Statistics, Newton, H.J. (ed), Vol 24, pp. 40-44, Mar.1992.

Lubars, M.D., "Wild-Spectrum Support for Software Reusability", RMISE Workshop on Software Reuse, Rocky Mountain Institute of Software Engineering, Boulder, Colo., pp. 14-15, Oct. 1987.

Maiden, N.A.M., "Analogy as A Paradigm for Specification Reuse", Software Engineering Journal, Vol. 6, No. 1, pp. 3-15, Jan. 1991.

Matsumoto, Y., "SWB system: A Software Factory", in Software Engineering Envirnments, Hunke, H.(ed.), New York: North-Holland, pp. 305-317, 1981.

Matsumoto, Y., "Some Experience in Promoting Resuable Software Presentation in Higher Abstract Levels", IEEE Trans. on Software Engineering, Vol. SE-10, N0. 5, pp. 502-513, Sep. 1984.

Matsumoto, Y., "A Software Factory: An Overall Approach to Software Production", in Software Resuability, Freeman, P. (ed), IEEE Computer Society Pree, Washington, D.C., pp.155-178, 1987.

McCain, R., "Reusable Software Component Construction: A Product-oriented Paradigm. In Proceedings of the 5th AIAA/ACM/NASA/IEEE Computers in Aerospace Conference, Long Beach, CA, pp. 125-135, 1985.

McCracken, D.D. and Jackson, M.A., "Life Cycle Concept Considered Harmful", ACM Software Engineering Notes, Vol. 7, No. 2, pp. 29-32, 1982.

McIlroy, M.D., "Mass-Produced Software Components", in Software Engineering Reports on a Conference Sponsored by the NATO Science Committee, Naur, P. and Randell, B. (eds.), Scientific Affairs Div., NATO, Brussels, pp.151-155, 1969.

Meyer, B., "Software Reusability: The Case for Object-oriented Design", IEEE Software, Vol. 4, No. 2, pp. 50-64, Mar. 1987.

Neighbors, J.M., "The Draco Approach to Constructing Software from Reusable Components", IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, pp. 564-574, 1984.

Neighbors, J.M., "Software Construction using Components", Ph. D. Thesis University of California, Irvine, Department of Information and Computer Science, p. 154, 1980.

Neighbors, J.M., "Draco: A Method for Engineering Reusable Software System", in Software Reusability: Volumn I, Concepts and Models, Biggerstaff, T.J. and Perlis, A.J., (eds.), ACM Press, Addison-Wesley Publishing, pp. 295-319, 1989.

Parnas, D.L., "A Technique for the Specification of Software Modules with Examples", Communications of ACM, Vol. 15, No. 5, pp. 330-336, 1972.

Prieto-Diaz, R., "Implementing Faceted Classification for Software Reuse", Communications of the ACM, Vol.34, No. 5, pp. 89-97, May 1991.

Prieto-Diaz, R., "A Software Classification Scheme", Ph. D. Thesis, Thesis University of California, Irvine, Department of Information and Computer Science, p. 194, 1985.

Prieto-Diaz, R., "Domain Analysis for Reusability", In Proceedings of COMPSAC'87, Tokyo, Japan, 1987.

Prieto-Diaz, R. and Jones, G.A., "Breathing New Life into Old Software" in Software Reuse: Emerging Technology, Trace, w. (ed), IEEE Computer Society Pree, Washington, D.C., pp. 153-160, 1988.

Royce, W.W., "Managing the Development of Large Software Systems: Concepts and Techniques", Proceedings WESCON, pp. 1-9, 1970.

Rumbaugh, J., Blaha, M, Premerlani, W., Eddy, F. and Lorensen, W., "Object-oriented modeling and design", Prentice-Hall, 1991.

Shlaer, S. and Mellor, S.J., "An Object-oriented Approach to Domain Analysis", ACM SIGSOFT Software Engineering Notes Vol. 14, No.5, pp.66-77, Jul. 1989.

Tajima, D. and Matsubara, T., "Inside the Japanese Software Indusstry", Computer, Vol. 17, No. 3, pp. 34-43, Mar. 1984.

Tanik, M.M. and Chan, E.S., "Fundamentals of Computing for Software Engineers", Van Nostrand Reinhold, New York, 1991.

Tracz, W., "Software Reuse Myths", ACM SIGSOFT Software Engineering Notes, Vol. 13, No. 1, pp. 17-21, Jan. 1988,

Tracz, W., "Software Reuse: Motivators and Inhibitors", Proceedings of COMPCONS'87, pp. 358-363, 1987.

Walker, I., "Requirements of an Object-oriented Design Method", Software Engineering Journal, Vol. 7, No. 2, pp. 102-113, 1992.

Wegner, P., "Capital-Intensive Software Technology", IEEE Software, Vol. 1, No. 3, pp. 7-32, Jul. 1984.

Wirfs-Brock, R.J. and Johnson, R.E., "Surveying Current Research in the Object-Oriented Desgin", Communication ACM, Vol. 33, No. 9, pp.104-124, Sep. 1990.

Yin, W.P., Tanik, M.M., and Yun, D.Y.Y. Software Design Representation: Design Object Descriptive Attribute Notation (DODAN), Proceedings of International Conference on Computer Languages, 1988.

Zave, P., "The Operational Versus the Conventional Approach to Software Development", Communications of the ACM, Vol. 27, No.2, pp. 104-118, Feb. 1984.